Thesis for the Degree of Licentiate of Engineering

Property-based testing for functional programs

Nicholas Smallbone

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2011

Property-based testing for functional programs
Nicholas Smallbone

Technical Report 76L
ISSN 1652-876X
Department of Computer Science and Engineering
Research group: Functional Programming

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2011

Abstract

This thesis advances the view that property-based testing is a powerful way of testing functional programs, that has advantages not shared by traditional unit testing. It does this by showing two new applications of property-based testing to functional programming as well as a study of the effectiveness of property-based testing.

First, we present a tool, QuickSpec, which attempts to infer an equational specification from a functional program with the help of testing. The resulting specifications can be used to improve your understanding of the code or as properties in a test suite. The tool is applicable to quite a wide variety of situations.

Second, we describe a system that helps to find race conditions in Erlang programs. It consists of two parts: a randomised scheduler to provoke unusual behaviour in the program under test and allow replayability of test cases, and a module that tests that all of the functions of an API behave atomically with respect to each other.

Finally, we present an experiment we carried out to compare property-based testing against test-driven development. The results were inconclusive, but in the process we developed a black-box algorithm for automatically grading student programs by testing, by inferring for each program a set of bugs that the program contains.

Contents

Acknowledgements

First, I would like to thank my supervisor, Koen Claessen, who has been a constant source of excellent discussions and has always managed to unstick me whenever I got stuck. My friends at the department, who make sure there is never a dull moment here; and colleagues, who make it such a pleasant place to be. Finally, I would like to thank my family for putting up with me and encouraging me all these years, especially my brilliant big brother.

Introduction

Much of the effort in developing software goes into finding and fixing bugs. A 2002 study [Tassey, 2002] found that software bugs cost the US economy $60 billion a year. Bugs are, as they have always been, a thorn in the side of software developers.

In traditional unit testing, we write down a set of test cases by hand that we hope will expose any bugs in the software. For example, a test suite for a stack might check that pushing an element onto the stack and then popping it doesn't alter the stack, with assertions such as:

```
pop (push 1 empty) == empty
pop (push 3 (push 2 empty)) == push 2 empty
pop (push -1 (push 3 (push 2 empty))) ==
  push 3 (push 2 empty)
```

In our opinion, writing test cases by hand like this has a number of drawbacks:

- Coming up with individual test cases is a chore.
- There is a high risk that we will forget to test some edge case or other.

  We might guard against this by using some coverage metric to check that the test suite exercises the program in enough ways. However, such metrics are only heuristics: a test suite with 100% coverage might still be missing an important test case, and code coverage can give a false sense of security [Marick, 1999].
- We should not add redundant test cases to our test suite, but it is not clear how to tell if a test case is redundant. In the example above, since we have a test case `pop (push 1 empty) == empty` we should probably not include the similar test case `pop (push 2 empty) == empty`, because both are likely to exercise the code in exactly the same way, and the only effect of including the second test case will be to make the author of the test suite type more.

  The author of the test suite has to use their ingenuity to notice when a test case is not needed, and if we mistakenly identify a test case as redundant then we will end up with an incomplete test suite and may miss a bug.

- If we want to have more confidence that the program is correct, it is not possible to just tell the computer to run more tests: we have to come up with and write down some more tests ourselves.

- Such test suites are of little use when testing concurrent programs, as we will see later on.

Practitioners of property-based testing argue that writing *properties* instead of test cases avoids all these problems, and we agree. The idea of property-based testing is to write a *specification* of the system under test in a suitable language, which is then tested using a large collection of test data invented by the computer.

In functional programming the most popular tool for property-based testing is QuickCheck [Claessen and Hughes, 2000], which we use throughout this thesis. The user writes properties that can universally quantify over the set of test data; QuickCheck then generates random test data to check the property. For example, we can test the stack example above by writing a Haskell predicate of type `Stack -> Elem -> Bool`:

```
prop stack elem = top (push elem stack) == stack
```

which corresponds to the logical property

$$\forall stack\, \forall elem\, \mathsf{top}(\mathsf{push}(elem, stack)) = stack$$

QuickCheck will test the property by generating random stacks and random things to push on them.

Because it automatically and unbiasedly generates test cases from a property, QuickCheck isn't prone to any of the problems we mentioned above: we can test the property as thoroughly as we want, and QuickCheck will not "forget" an edge case like a human tester might.

However, QuickCheck is more than just a tool for generating test cases:

1. QuickCheck properties are really *specifications*, which can be tested but just as well model-checked [QCM] or, in principle, given to a theorem prover [Cov]. Thus property-based testing can help to bridge the gap between formal verification and unit testing. We do not explore this further in the thesis, though.

2. QuickCheck properties are really *programs*, so we have the ability to write more sophisticated tests than are common in traditional unit testing. Using the terminology of software testing, a QuickCheck property is a *test oracle* that has the freedom to decide whether or not the test case passed.

   This helps greatly with testing nondeterministic programs, where it's not possible to write down a single correct answer for each test case. Our paper on finding race conditions relies on this ability, which is not really present with "traditional" unit testing.

   A recent paper not involving the author of this thesis [Hughes et al., 2010] uses the power of QuickCheck properties to find concurrency bugs in

`ejabberd`, a popular instant messaging server. The kind of bugs found can not really be tested for using traditional test cases, because it's necessary to analyse the test results extensively to tell if the test passed or failed.

The main aim of this thesis is to show that property-based testing has real extra power over traditional unit testing when it comes to testing programs, and is not just a convenient way to avoid writing test cases.

Random testing or exhaustive testing? QuickCheck generates its test data at random: rather a scattershot approach to testing. This is frowned upon by software testing people, who tend to prefer a more systematic approach to test data generation.

The tool *SmallCheck* uses a property language similar to QuickCheck but generates test data exhaustively, up to a certain size bound, instead of at random. This has the advantage that you have a stronger guarantee of what has been tested once your property passes. It also becomes possible to extend the property language by adding existential quantification, which is not really desirable in QuickCheck: we would not really learn anything if a property $\forall x \exists y P(x, y)$ failed, because it might just mean that we weren't able to guess the value of the witness y; with SmallCheck, we know that there *is* no witness up to the size bound.

It is not really clear if one method is better. Exhaustive testing is more predictable, but suffers from exponential blowup: we can only try small test cases. By constrast, with random testing we can try much bigger test cases but can't explore the space of test cases as thoroughly. If a property only has one counterexample then exhaustive testing is the best way to find it; but, normally there is enough symmetry in the problem that if there is one counterexample to a property then there are many similar counterexamples, in which case random testing works well. With exhaustive testing we need to be careful not to generate too many redundant test cases, just like when testing by hand, but with random testing there is no harm unless the redundancy disturbs the distribution of test data: we are just *sampling* the set of test data and it makes no difference how big that set is.

Hamlet [2006] studies the advantages and disadvantages of random testing and concludes that, contrary to popular opinion, there are occasions when random testing gives the *best* test data. The examples he points out are when the input domain is large and unstructured, and when generating a sequence of imperative commands to be executed. Ciupa et al. [2007] shows experimentally that random testing works fine on object-oriented software, and that the worries that random testing is too indiscriminate are unfounded.

Functional programming and testing We agree with the widely-held belief that functional programs ought to be easier to test than imperative ones:

- They are easier to reason about and specify, and this advantage carries over directly when testing functional programs using properties, which are after all specifications.

- They tend to be more modular, as Hughes [1989] argues, so we ought to be able to test them at a finer-grained level.

A recurring theme of the papers in this thesis is that *avoiding side effects* has serendipitous effects on testing. As discussed below, the tools in the first two papers are very much intended for a programming style with sparse side effects—do the same thing in an imperative language, and the tools would lose a lot of their power.

Paper I: QuickSpec: Formal Specifications for Free!

It is all very well to trumpet the merits of property-based testing, but real code often comes with no test-suite at all, let alone a nice set of properties. QuickSpec is our tool that takes a side-effect-free module of a functional program and, by testing it, produces *equations* that describe the functions of that module, which can be turned into properties for testing or just used as an aid for understanding the module.

For example, Hughes [1995] describes a library for pretty-printing that includes the following functions:

```
($$), (<>) :: Layout -> Layout -> Layout
nest :: Int -> Layout -> Layout
text :: [Char] -> Layout
```

A `Layout` is roughly a combination of text to be printed and layout information. The `text` function turns any string into a trivial `Layout`; the `nest` function takes any `Layout` and causes it to be indented when it's printed; the `$$` operator combines two `Layout`s vertically, placing one on top of the other, whereas `<>` will put one `Layout` next to the other, on the same line. The pretty-printing library also includes a `sep` operator, which combines two `Layout`s either horizontally or vertically: horizontally if there is enough space to do so, vertically otherwise. This function is really where the power of the library comes from, but we leave it out here for simplicity.

To run QuickSpec, we simply give the list of functions we want to test, in this case the ones above. We also give QuickSpec any auxiliary functions and constants that we think may be useful for specifying the API:

```
0 :: Int
(+) :: Int -> Int -> Int
"" :: [Char]
(++) :: [Char] -> [Char] -> [Char]
```

In the current implementation, we also have to give a collection of variables that the equations may quantify over:

```
i, j, k :: Int
x, y, z :: Elem
d, e, f :: Layout
s, t, u :: [Char]
```

Once we have done this, QuickSpec finds the following equations (and no others), all tested on several hundred randomly-generated test cases:

```
1: nest 0 d == d
2: nest j (nest i d) == nest i (nest j d)
3: d<>nest i e == d<>e
4: nest (i+j) d == nest i (nest j d)
5: (d$$e)$$f == d$$(e$$f)
6: nest i d<>e == nest i (d<>e)
7: (d$$e)<>f == d$$(e<>f)
8: (d<>e)<>f == d<>(e<>f)
9: nest i d$$nest i e == nest i (d$$e)
10: text s<>text t == text (s++t)
11: d<>text "" == d
```

These laws give us some insight into the behaviour of the pretty-printing library. For example, law 3 tells us that horizontally composing two layouts, `d<>e`, ignores the indentation level of `e`—in other words, only the indentation of the *first* thing on each line counts. Law 9 tells us that when indenting a multi-line layout, each line is individually indented. (If only the first line were indented, we would instead see the law `nest i d$$e == nest i (d$$e)`.) Line 10 tells us that the characters in a string are composed horizontally when we use `text`.

In his paper, Hughes gives a list of 11 axioms that characterise the pretty-printing combinators. 10 of these 11 are also found by QuickSpec! (The 11th, which gives conditions when it is possible to push a `<>` inside a `$$`, is a little too complicated for QuickSpec to find.) So QuickSpec redeems itself well in this case when compared to a manually-written specification.

Related work Henkel et al. [2007] describe a similar system for Java programs. Their system also uses testing to generate equations that seem to hold for a collection of functions. The existence of such a system might be surprising: after all, equations seem ill-suited to reasoning about imperative programs, which are normally specified using pre- and postconditions.

Henkel et al. work around this by only dealing with programs that they know how to model functionally. They do this by imposing a syntactic restriction on the program fragments that can appear in their equations: the only thing a program fragment can do is to invoke several methods of a single object, one after the other. The arguments to the method calls must be atomic—variables or constants.

This restriction means that each program fragment will only mention one object, which neatly rules out aliasing. Because of this, they can model their stateful methods by pure functions taking the "old" object to the "new" object. This allows them to safely use equational reasoning.

Unfortunately, this restriction severely limits the scope of their tool: they are not able to deal with nested method calls (calling a method and using the result as the argument to another method) or indeed anything other than a flat sequence of method calls. Similarly, they do not support binary operators: if testing, say, a "set" class with their tool, they can find laws about `insert`

and `delete` but not `union`, since this will take a set as a parameter and their tool can only deal with programs that mention just *one* set. Testing something such as the pretty-printing library above is out of the question.

It is hard to see how to fix their tool to remove these restrictions.

Side effects are trouble All of their problems come from the fact that their tool has to deal with side effects, which is unavoidable since a Java program consists of nothing *but* side effects chained in the right order. By contrast, large parts of functional programs are side-effect-free—even in an impure language—so that we are able to completely disallow side effects in our equations and still have a convincing tool.

We have considered adding support for code with monadic side effects to QuickSpec, but it would indeed cause lots of complications. We would need support for λ-abstraction so that we could have equations containing monadic bind. (We've attempted to represent program fragments without using λ-expressions but the resulting equational theory doesn't seem to be strong enough.) Aliasing might also cause trouble, although there is an equational specification for mutable references that might help [Plotkin and Power, 2002]. What is clear is that this kind of tool is much more plausible in a functional setting than an imperative one.

Paper II: Finding Race Conditions in Erlang with QuickCheck and PULSE

Our second paper concerns finding race conditions in Erlang programs.

In our opinion, writing single test cases by hand is hopeless when it comes to testing concurrent programs for race conditions, for two reasons:

1. Concurrent programs tend to be highly nondeterministic. Thus a single test case may have several plausible correct outcomes, depending on what order everything happened to execute in. The larger the test case, the more possible outcomes, and there is often no obvious way to systematically enumerate those outcomes. Therefore, it can be hard even to say if a test passed or failed.

2. Race conditions are famously hard to reproduce, and may reveal themselves only under very particular timing conditions. In order for a test case to provoke such a race condition, you often need to "massage" the test case by inserting carefully-chosen delays. Such test cases depend heavily on the exact timing behaviour of the code under test and may become ineffective if you even make the code a little faster or slower—hardly a basis for a good test suite!

   When debugging a race condition, you might want to add print statements to see what is going on—but that often has the effect of skewing the timing and making the race condition vanish again!

These problems make testing concurrent programs in a traditional way a nightmare. We present an approach based around QuickCheck to alleviate the

problems. Our approach has two parts, `eqc_par_statem` (short for "Erlang QuickCheck parallel state machine") to help in writing the tests and PULSE to make test cases repeatable.

`eqc_par_statem` The idea behind `eqc_par_statem` is to test for one particular property of a concurrent API: namely, that all the functions of that API behave atomically. This is by no means the *only* property we might want to test of a concurrent API, but is normally a desirable property: after all, a concurrent API whose operations *don't* behave atomically will tend to drive its users mad.

How do we test for atomicity? First, the user must supply a *sequential* specification of the API, giving preconditions, postconditions etc. for each operation. Erlang QuickCheck already supports these specifications for the purposes of testing sequential imperative code [Arts et al., 2006].

Using the sequential specification we can generate processes that invoke the API. We generate a pair of such processes; they consist of a sequence of API commands. Then we run the two processes in parallel and observe the results. Finally, we have to check if in this *particular* run of the system, the API functions behaved atomically.

We do this by trying to *linearise* [Lamport, 1979] the API calls that were made: we search for an interleaving of the two command sequences that would give the same results that we actually observed. In other words, we work out if the system behaved *as if* only one command was running at a time. If there is no such interleaving, then the only possible explanation is that the two processes interfered with each other, and we report an error.

PULSE The second component of our approach is PULSE, a replacement for the standard Erlang scheduler.

The standard Erlang scheduler suffers from two problems when testing concurrent code:

- It leads to nondeterministic behaviour: running the same test several times may result in a different outcome each time, because the order that processes run in is partly a result of chance. This leads to unrepeatable test cases.
- Paradoxically, it is also *too deterministic*: the Erlang scheduler preempts processes at regular intervals, so that each test case will have a similar effect each time we run it. This might result in the system behaving differently when idle and when under load, for example. Thus a property we test may be *false* but might never be *falsified* if we do not test it in exactly the right circumstances.

PULSE is intended to take over scheduling decisions from the standard Erlang scheduler. Whenever there is a choice to be made—which process should execute next, for example—it chooses randomly. This exposes as wide a range of behaviour as possible from the program under test.

PULSE also *records* the choices it made into a log. Should we wish to repeat a test case, we can give the log to PULSE and it will make the same scheduling choices again, thus giving us repeatability. We can also turn this log into a graphical trace of the system's behaviour, to help with debugging.

Case study We applied our tools to a piece of industrial Erlang code with a mysterious race condition, which is described in the paper. We indeed found the race condition; unfortunately, it turned out to be a subtle design fault and impossible to fix without redesigning the API!

Related work It should be no surprise that researchers in imperative programming have already worked on tools to make debugging of concurrent programs more tractable. Related work includes Chess for .NET [Musuvathi et al., 2008] and RaceFuzzer for Java [Sen, 2008b].

When comparing PULSE to these tools, the most surprising thing is how much simpler our approach is. We speculate that this is due to the fact that Java and .NET use shared memory concurrency. Because of this, almost any line of code can potentially participate in a race: every operation has an effect that can be observed by other processes, making the problem of finding the real race conditions harder. By contrast, Erlang uses message passing concurrency and only provides shared state in the form of a mutable map library, `ets`. PULSE only needs to preempt a process when it sends a message or performs a genuine side effect, rather than on every variable access, so there are far fewer ways to schedule any program and PULSE has an easier time finding the schedules that provoke bugs. In other words, just as with QUICKSPEC, we gain an advantage by working in a setting where side effects are only used when actually useful.

The other main difference between our work and that for Java and .NET is that they don't have any equivalent of `eqc_par_statem` for atomicity testing. Chess requires the user to state a property they want to hold: for example, that the system must never deadlock. RaceFuzzer checks that there are no data races: where one thread accesses a variable at the same time as another thread is writing to it. However, since this is a very low-level property, sometimes harmless data races are found.

PULSE versus model checking Model checking can also be used to provoke race conditions: a model checker will systematically enumerate all possible schedules of a system, so guaranteeing to find a race condition in a particular test case if one exists. In fact, Chess is designed as a model checker, and there is already a model checker for Erlang [Fredlund and Svensson, 2007]. So why do we need PULSE at all? The answer is that model checking suffers from exponential blowup. Although when it does work we get a stronger guarantee than PULSE, for larger examples using a model checker is just infeasible but PULSE will work perfectly happily. The tension is similar to that between random and exhaustive test data generation that was mentioned earlier.

Paper III: Ranking Programs using Black Box Testing

Empirical studies have demonstrated that test-driven development can be more effective than test-after methods [Canfora et al., 2006],

We ran an unsuccessful experiment hoping to compare so-called "property-driven development" against test-driven development: do you get better results from test-driven development if you write properties instead of individual test cases? Since property-based testing is itself quite a niche area, we didn't find any previous experiment testing this. We took a group of 13 students and got each of them to solve some programming problems and test their code as they went: half using QuickCheck and the other half using HUnit, a traditional unit testing tool. Afterwards, we analysed the results to try to answer two questions:

1. Did the programs that were tested with QuickCheck work better than the ones that were tested with HUnit?

2. Were the QuickCheck test suites more effective at finding bugs than the HUnit test suites?

It quickly became apparent that the number of subjects who took part in the experiment (13) was too small for us to draw any firm conclusions. For the first question the situation was entirely hopeless. When considering the second question we did spot something curious: half of the QuickCheck test suites were worthless (they failed to detect a single bug in any student's submission) but the other half were among the *best* test suites submitted—they were as good as or better than all of the HUnit test suites. All of the students using HUnit missed some edge case or other, whereas the QuickCheck properties, although harder to write in the first place, did not have any "blind spots". However, the sample size was too small for us to judge whether this is the case in general. We hope, however, to perform a larger-scale experiment in the future that will fix the flaws in this one.

The ranking algorithm When marking the students' submissions we were wary of introducing bias in the scores, which led to the real contribution of the paper: an algorithm to automatically "grade" a collection of programs.

At our disposal we have the student submissions, all trying to implement the same specification, and a QuickCheck property to test an implementation's correctness. We might think to just generate a lot of test cases and count how many each submission passes to get a score for the submission, but this introduces a possible bias: the exact *distribution of test data* we use will affect the scores given to the students.

Instead, we found a way to infer a set of *bugs* from the test results. We assume that each test case provokes some number of bugs, each program has some set of bugs, and a program will fail a test case if the test case provokes a bug and the program has that bug. We show in the paper how to infer a relation between test cases and bugs, satisfying the requirements above, that gives us the smallest number of bugs. Finally, the score of each program is simply the number of bugs in our inferred set that the program doesn't have.

We found that the set of bugs inferred does not depend much on the particular test data generated, a good sign. The algorithm only works because we are able to generate a huge amount of test data: if we only had a manually-written test suite, the choice of test suite would bias the scores; for example, if the test suite was not complete then some bugs would not be found at all!

We found that the set of bugs inferred by our algorithm was not exactly the same as the set we intuitively would choose. This happened because some bugs interfere: a test case may provoke two bugs A and B in such a way that a program with bug A or bug B will fail the test, but a program with both bugs will somehow pass it! Our algorithm will then count the test case as a bug in its own right. This leads to a curious observation: this test case can detect bug A, but only in programs that do not have bug B. If you are writing a test suite for your program, including this test will give you a *false sense of security*—you may think you have a test case for bug A, but it might not always work! We came across more than one such test case, and they all looked like reasonable tests—a reason to be suspicious of handwritten test cases!

Conclusions

We hope that this thesis supports the view that property-based testing is a powerful way to test programs, that it can test things that traditional unit testing cannot easily, and that it applies especially well to functional programs. The contributions of this thesis are:

- A tool that *automatically infers* specifications from pure functional programs.
- A tool that provokes and then flags race conditions in concurrent functional programs, a task which seems impossible to do without the help of properties.
- An algorithm for automatically grading a large set of programs (such as student assignments). Unlike the first two contributions, this works even if the programs tested are not functional, but it does rely on having a specification and the ability to generate large amounts of test data.

The development of the last algorithm arose from an unsuccessful attempt to determine experimentally how well property-based testing works, which we hope might form the basis of a future experiment along the same lines.

Paper I

QuickSpec: Formal Specifications for Free!

This paper was originally presented at TAP 2010 in Malaga, under the title "QuickSpec: Guessing Formal Specifications using Testing". This is an extended version which has been submitted to a special issue of the Software Quality Journal.

QuickSpec: Formal Specifications for Free!

Koen Claessen, Nicholas Smallbone and John Hughes

Abstract

We present QuickSpec, a tool that automatically generates algebraic specifications for sets of pure functions. The tool is based on testing, rather than static analysis or theorem proving. The main challenge QuickSpec faces is to keep the number of generated equations to a minimum while maintaining completeness. We demonstrate how QuickSpec can improve one's understanding of a program module by exploring the laws that are generated using two case studies: a heap library for Haskell and a fixed-point arithmetic library for Erlang.

1 Introduction

Understanding code is hard. But it is vital to understand what code does in order to determine its correctness.

One way to understand code better is to write down one's expectations of the code as formal specifications, which can be tested for compliance, by using a property-based testing tool. Our earlier work on the random testing tool QuickCheck [Claessen and Hughes, 2000] follows this direction. However, coming up with formal specifications is difficult, especially for untrained programmers. Moreover, it is easy to forget to specify certain properties.

In this paper, we aim to aid programmers with this problem. We propose an automatic method that, given a list of function names and their object code, uses testing to come up with a set of *algebraic equations* that seem to hold for those functions. Such a list can be useful in several ways. Firstly, it can serve as a basis for documentation of the code. Secondly, the programmer might gain new insights by discovering new laws about the code. Thirdly, some laws that one expects might be missing (or some laws might be more specific than expected), which points to to a possible mistake in the design or implementation of the code.

Since we use testing, our method is potentially unsound, meaning some equations in the list might not hold; the quality of the generated equations is only as good as the quality of the used test data, and care has to be taken. Nonetheless, we still think our method is useful. However, our method is still complete in a precise sense; although there is a limit on the complexity of the expressions that occur in the equations, any syntactically valid equation that actually holds for the code can be derived from the set of equations that QuickSpec generates.

Our method has been implemented for the functional languages Haskell and Erlang in a tool called QuickSpec. At the moment, QuickSpec only works for purely functional code, i.e. no side effects. (Adapting it to imperative and other side-effecting code is ongoing work.)

1.1 Examples

Let us now show some examples of what QuickSpec can do, by running it on different subsets of the Haskell standard list functions. When we use QuickSpec, we have to specify the functions and variable names which may appear in equations, together with their types. For example, if we generate equations over the list operators

```
(++) :: [Elem] -> [Elem] -> [Elem]     -- list append
(:)  :: Elem -> [Elem] -> [Elem]       -- list cons
[]   :: [Elem]                         -- empty list
```

using variables `x,y,z :: Elem` and `xs,ys,zs :: [Elem]`, then QuickSpec outputs the following list of equations:

```
xs++[] == xs
[]++xs == xs
(xs++ys)++zs == xs++(ys++zs)
(x:xs)++ys == x:(xs++ys)
```

We automatically discover the associativity and unit laws for append (which require induction to prove). These equations happen to comprise a complete characterization of the `++` operator. If we add the list reverse function to the mix, we discover the additional familiar equations

```
reverse [] == []
reverse (reverse xs) == xs
reverse xs++reverse ys == reverse (ys++xs)
reverse (x:[]) == x:[]
```

Again, these laws completely characterize the `reverse` operator. Adding the `sort` function from the standard `List` library, we compute the equations

```
sort [] == []
sort (reverse xs) == sort xs
sort (sort xs) == sort xs
sort (ys++xs) == sort (xs++ys)
sort (x:[]) == x:[]
```

The third equation tells us that `sort` is idempotent, while the second and fourth strongly suggest (but do not imply) that the result of `sort` is independent of the order of its input.

Adding the `usort` function (equivalent to `nub . sort`), which sorts and eliminates duplicates from its result, generates the same equations together with one new one:

```
usort (xs++xs) == usort xs
```

which strongly suggests that the result of `usort` is independent of repetitions in its input.

If we add a `merge` function for ordered lists, then we obtain equations relating `merge` and `sort`:

```
merge [x] (sort xs) == sort (x:xs)
merge (sort xs) (sort ys) == sort (xs++ys)
```

We also obtain other equations about `merge`, such as the somewhat surprising

```
merge (xs++ys) xs == merge xs xs++ys
```

Note that this holds even for *unordered* `xs` and `ys`, but is an artefact of the precise definition of `merge`.

We can deal with higher-order functions as well. Adding the function `map` together with a variable `f :: Elem -> Elem`, we obtain:

```
map f [] == []
map f (reverse xs) == reverse (map f xs)
map f xs++map f ys == map f (xs++ys)
f x:map f xs == map f (x:xs)
```

Because our signature concerns sorting and sorted lists, there may be laws about `map f` that only hold when `f` is monotone. Although QuickSpec does not directly support such *conditional laws*, we can simulate them by adding a new type `Monotonic` of monotone functions. Given an operator that applies a monotone function to its argument...

```
monotonic :: Monotonic -> Elem -> Elem
```

...and a variable `f :: Monotonic`, we obtain

```
map (monotonic f) (sort xs) == sort (map (monotonic f) xs)
merge (map (monotonic f) xs) (map (monotonic f) ys) ==
  map (monotonic f) (merge xs ys)
```

The latter equation is far from obvious, since it applies even to *unordered* `xs` and `ys`.

All of the above uses of QuickSpec only took a fraction of a second to run, and what is shown here is the verbatim output of the tool.

1.2 Queues

The Erlang standard libraries include an abstract datatype of *double-ended queues*, with operations to add and remove elements at the left and the right ends, to join and reverse queues, and so on. The representation is the well-known one using a pair of lists, which gives amortized constant time for many operations [Burton, 1982]. Running QuickSpec on a signature including

```
new() -> queue()
tail(queue()) -> queue()
liat(queue()) -> queue()
reverse(queue()) -> queue()
in(elem(),queue()) -> queue()
in_r(elem(),queue()) -> queue()
join(queue(), queue()) -> queue()
to_list(queue()) -> list()
```

with the variables

```
X, Y, Z :: elem()
Q, Q2, Q3 :: queue()
```

we obtain equations such as

```
join(new(),Q) == Q
join(Q,new()) == Q
join(join(Q,Q2),Q3) == join(Q,join(Q2,Q3))
to_list(Q) ++ to_list(Q2) == to_list(join(Q,Q2))
```

which tell us that the `join` operator is very well-behaved, and

```
reverse(in(X,Q)) == in_r(X,reverse(Q))
tail(reverse(Q)) == reverse(liat(Q))
```

which relate the insertion and removal operations at each end of the queue to each other in a pleasing way. On the other hand, we also obtain

```
to_list(Q) ++ [X] == to_list(in(X,Q))
[X|to_list(Q)] == to_list(in_r(X,Q))
```

which reveal that if we thought that `in_r` inserted an element at the *right* or the *rear* of the queue, then we were wrong! The `in` function inserts elements on the right, while `in_r` inserts them on the left, of course.

However, some of the generated equations are more intriguing. Consider this one:

```
tail(in_r(Y,Q)) == tail(in_r(X,Q))
```

It is not unexpected that this equation holds—it says that adding an element to the front of a queue, then removing it again, produces a result that does not depend on the element value. What *is* unexpected is that our tool does not report the simpler form:

```
tail(in_r(X,Q)) == Q
```

In fact, the reason that we do not report this simpler equation is that it is not true! One counterexample is `Q` taken to be `in(0,new())`, which evaluates to `{[0],[]}`, but for which the left-hand side of the equation evaluates to `{[],[0]}`. These are two different representations of the "same" queue, but because the representations do differ, then by default QuickSpec considers the equation to be false.

We can also tell QuickSpec to compare the contents of the queues rather than the representation, in which case our equation becomes true and we get a slightly different set of laws.

1.3 Arrays

In 2007, a new library was added to the Erlang distribution supporting purely functional, flexible arrays, indexed from zero [Carlsson and Gudmundsson,

2007]. We applied QuickSpec to subsets of its API. Using the following signature,

```
new() -> array()
get(index(),array()) -> elem()
set(index(),elem(),array()) -> array()
default_element() -> elem()
```

with variables

```
X, Y, Z :: elem()
I, J, K :: index()
A, B, C :: array()
```

we obtained these laws:

```
get(I,new()) == default_element()
get(I,set(I,X,A)) == X
get(I,set(J,default_element(),new())) == default_element()
get(J,set(I,X,new())) == get(I,set(J,X,new()))
set(I,X,set(I,Y,A)) == set(I,X,A)
set(J,X,set(I,X,A)) == set(I,X,set(J,X,A))
```

The `default_element()` is not part of the arrays library: we introduced it and added it to the signature after QuickSpec generated the equation

```
get(I,new()) == get(J,new())
```

Since the result of reading an element from an empty array is constant, we might as well give it a name for use in other equations. When we do so, then the equation just above is replaced by the first one generated.

Some of the equations above are very natural: the second says that writing an element, then reading it, returns the value written; the fifth says that writing to the same index twice is equivalent to just writing the second value. The sixth says that writing the *same* value `X` to two indices can be done in either order—but why can't we swap *any* two writes, as in

```
set(J,Y,set(I,X,A)) =?= set(I,X,set(J,Y,A))
```

The reason is that this equation holds only if `I /= J` (or if `X == Y`, of course)! It would be nice to generate *conditional equations* such as

```
I /= J ==> set(J,Y,set(I,X,A)) == set(I,X,set(J,Y,A)).
```

Presently we have a prototype version of QuickSpec that can generate such laws (and indeed generates the one above), but it is rather preliminary and there are several creases to be ironed out yet. The version of QuickSpec that we discuss in this paper doesn't generate conditional laws.

The fourth equation

```
get(J,set(I,X,new())) == get(I,set(J,X,new()))
```

is a little surprising at first, but it does hold—either both sides are the default element, if `I` and `J` are different, or both sides are `X`, if they are the same.

Finally, the third equation is quite revealing about the implementation:

```
get(I,set(J,default_element(),new())) == default_element()
```

A new array contains the default element at every index; evidently, setting an index explicitly to the default element will not change this, so it is no surprise that the `get` returns this element. The surprise is that the second argument of `get` appears in this complex form. Why is it `set(J,default_element(),new())`, rather than simply `new()`, when both arrays have precisely the same elements? The answer is that *these two arrays have different representations*, even though their elements are the same. That the equation appears in this form tells us, indirectly, that

```
set(J,default_element(),new()) /= new()
```

because if they were equal, then QuickSpec would have simplified the equation. In fact, there is another operation in the API, `reset(I,A)`, which is equivalent to setting index `I` to the default element, and we discover in the same way that

```
reset(J,new()) /= new()
```

`set` and `reset` could have been defined to leave an array unchanged if the element already has the right value—and this could have been a useful optimization, since returning a different representation forces `set` and `unset` to copy part of the array data-structure. Thus this *missing equation* reveals a potentially questionable design decision in the library itself. This is exactly the kind of insight we would like QuickSpec to provide!

The arrays library includes an operation to *fix* the size of an array, after which it can no longer be extended just by referring to a larger index. When we add `fix` to the signature, we discover

```
fix(fix(A)) == fix(A)
get(I,fix(new())) == undefined()
set(I,X,fix(new())) == undefined()
```

Fixing a fixed array does not change it, and if we fix a new array (with a size of zero), then any attempt to get or set an element raises an exception[1].

When we include the array resizing operation, we obtain a number of new laws:

```
get(I,resize(J,new())) == default_element()
get(J,resize(J,A)) == get(I,resize(I,A))
resize(I,resize(I,A)) == resize(I,A)
resize(J,fix(C)) == fix(resize(J,C))
set(I,X,resize(I,new())) == set(I,X,new())
```

The first reveals, indirectly, that

[1] We consider all terms which raise an exception to be equal—and `undefined()` is a built-in-to-QuickSpec term that always does so.

```
resize(J,new()) /= new()
```

which is perhaps not so surprising. The second equation is interesting: it holds because the `I`'th index is just beyond the end of `resize(I,A)`, and so the result is either the default element (if `A` is a flexible array), or `undefined()` (if `A` is fixed). The equation tells us that which result we get depends only on `A`, not on `I` or `J`. It also tells us that the result is not *always* the default element, for example, since if it were, then we would have generated an equation with `default_element()` as its right-hand side.

The reader may care to think about why the third equation specifies only that two resizes *to the same size* are equivalent to one, rather than the more general (but untrue)

```
resize(I,resize(J,A)) =?= resize(I,A)
```

The fourth equation tells us that resizing and fixing an array commute nicely, while the fifth gives us a clue about the reason for the behaviour of `set` discussed earlier: setting an element can clearly affect the *size* of an array, as well as its elements, which is why setting an element to its existing value cannot always be optimized away.

1.4 Main related work

The existing work that is most similar to ours is Henkel et al. [2007]. They describe a tool for discovering algebraic specifications from Java classes using testing, using a similar overall approach as ours (there are however important technical differences discussed in the related work section later in the paper). However, the main difference between our work and theirs is that we generate equations between *nested expressions* consisting of functions and variables whereas they generate equations between Java program fragments that are *sequences of method calls*. The main problem we faced when designing the algorithms behind QuickSpec was taming the explosion of equations generated by operators with structural properties, such as associativity and commutativity, equations that are not even expressible as equations between sequences of method calls (results of previous calls cannot be used as arguments to later ones).

1.5 Contributions

We present a efficient method, based on testing, that automatically computes algebraic equations that seem to hold for a list of specified pure functions. Moreover, using two larger case studies, we show the usefulness of the method, and present concrete techniques of how to use the method effectively in order to understand programs better.

2 How QuickSpec Works

The input taken by QuickSpec consists of three parts:

- the compiled program,
- a list of functions and variables, together with their types, and
- test data generators for each of the types of which there exists at least one variable.

2.1 The method

The method used by QuickSpec follows four distinct steps:

1. We first generate a (finite) set of terms, called the *universe*, that includes any term that might occur on either side of an equation.
2. We use testing to partition the universe into *equivalence classes*; any two terms in the same equivalence class are considered equal after the testing phase.
3. We generate a list of *equations* from the equivalence classes.
4. We use *pruning* to filter out equations that follow from other equations by equational reasoning.

In the following, we discuss each of these steps in more detail, plus some refinements and optimisations to the basic method. As a running example we take a tiny signature containing the boolean operator `&&` and the constant `false`, as well as boolean variables `x` and `y`.

2.2 The universe

First, we need to pin down what kind of equations QuickSpec should generate. To keep things simple and predictable, we only generate one finite set of terms, the *universe*, and our equations are simply pairs of terms from the universe. Any pair of terms from the universe can form an equation, and both sides of an equation must be members of the universe.

What terms the universe should contain is really up to the user; all we require is that the universe be subterm-closed. The most useful way of generating the universe is letting the user specify a *term depth* (usually 3 or 4), and then simply produce all terms that are not too deep. The terms here consist of the function symbols and variables from the specified API.

The size of the universe is typically 1000 to 50,000 terms, depending on the application.

To model *exceptions*, our universe also contains one constant `undefined` at each type. The behaviour of `undefined` is to throw an exception when you evaluate it. This means that we can also find equations of the form `t == undefined`, which is true if `t` always throws an exception.

In our tiny boolean example, supposing that our maximum term depth is 2, the universe consists of the following terms (we leave out `undefined` for clarity):

```
x
y
false
x&&x
x&&y
x&&false
y&&x
y&&y
y&&false
false&&x
false&&y
false&&false
```

2.3 Equivalence classes

The next step is to gather information about the terms in the universe. This is the only step in the algorithm that uses testing, and in fact makes use of the program. Here, we need to determine which terms seem to behave the same, and which terms seem to behave differently. In other words, we are computing an equivalence relation over the terms.

Concretely, in order to compute this equivalence relation, we use a refinement process. We represent the equivalence relation as a set of equivalence classes, partitions of the universe. We start by assuming that all terms are equal, and put all terms in the universe into one giant equivalence class. Then, we repeat the following process: We use the test data generators to generate test data for each of the variables occurring in the terms. We then refine each equivalence class into possibly smaller ones, by evaluating all the terms in a class and grouping together the ones that are still equal, splitting the terms that are different. The process is repeated until the equivalence relation seems "stable"; if no split has happened for the last 200 tests. Note that equivalence classes of size 1 are trivial, and can be discarded; these contain a term that is only equal to itself.

The typical number of non-trivial equivalence classes we get after this process lies between 500 and 5,000, again depending on the size of the original universe and the application.

Once these equivalence classes are generated, the testing phase is over, and the equivalence relation is trusted to be correct for the remainder of the method.

For our tiny example, the non-trivial equivalence classes are as follows:

```
{x, x&&x}
{y, y&&y}
{false, false&&x, false&&y, x&&false, y&&false,
 false&&false}
{x&&y, y&&x}
```

2.4 Equations

From the equivalence classes, we can generate a list of equations between terms. We do this by picking one representative term r from the class, and producing the equation $t = r$ for all other terms from the class. So, an equivalence class of size k produces $k-1$ equations. The equations look nicest if r is the *simplest* element of the equivalence class (according to some simplicity measure based on for example depth and/or size), but which r is chosen has no effect on the completeness of the algorithm.

However, this is not the full story. Taking our tiny boolean example again, we can read off the following equations from the equivalence relation:

```
1. x&&x == x
2. y&&y == y
3. x&&y == y&&x
4. x&&false == false
5. y&&false == false
6. false&&x == false
7. false&&y == false
8. false&&false == false
```

This is certainly *not* what we want to present to the user: there is a mass of redundancy here. Laws 2, 5 and 7 are just renamings of laws 1, 4 and 6; moreover, laws 1, 3 and 4 together imply all the other laws. Our eight equations could be replaced by just three:

```
1. x&&x == x
2. x&&y == y&&x
3. x&&false == false
```

Whittling down the set of equations is the job of the *pruning* step.

2.5 Pruning

Pruning filters out redundant laws from the set of equations generated above, leaving a smaller set that expresses the same information. That smaller set is what QuickSpec finally shows to the user.

In a real example, the number of equations that are generated by the testing phase can lie between 1,000 and 50,000. A list of even 1,000 equations is absolutely not something that we want to present to the user; the number of equations should be in the tens, not hundreds or thousands. Therefore, the pruning step is crucial.

Which laws are kept and which are discarded by our current implementation of QuickSpec is in some ways an arbitrary choice, but our choice is governed by the following four principles:

1. **Soundness**: we can only remove a law if it can be derived from the remaining laws. In other words, from the set of equations that our pruning

algorithm keeps we should be able to derive all of the equations that were deleted.

2. **Conciseness**: we should remove all obvious redundancy, for a suitably chosen definition of redundant.

3. **Implementability**: the method should be implementable and reasonably efficient, bearing in mind that we may have thousands of equations to filter.

4. **Predictability**: the user should be able to draw conclusions from the absence or presence of a particular law, which means that no ad-hoc decisions should be made in the algorithm.

We would like to argue that the choice of which laws to discard must *necessarily* be arbitrary. The "ideal" pruning algorithm might remove all redundant laws, leaving a minimal set of equations from which we can prove all the rest. However, this algorithm *cannot exist*, because:

- Whether an equation is redundant is in general undecidable, so a pruning algorithm will only be able to detect certain classes of redundancy and will therefore print out more equations than we would like.

- There will normally not be one unique minimal set of equations. In our boolean example, we could take out `x&&false == false` and replace it by `false&&x == false` and we would still have a minimal set. Any algorithm must make an arbitrary choice between these two sets.

There is one more point. We do not necessarily *want* to produce an absolutely minimal set of equations, even if we could. Our tool is intended for program understanding, and we might want to keep a redundant equation if it might help understanding. This is another arbitrary choice any pruning algorithm must make. For example, in the domain of boolean expressions, from the following laws...

```
x||true == true
x&&true == x
x&&(y||z) == (x&&y)||(x&&z)
```

...we can actually prove idempotence, `x||x == x`! (Take the third law and substitute `true` for `y` and `z`.) So a truly minimal set of equations for booleans would not include the idempotence law. However, it's quite illuminating and we probably do want to include it.

In other words, we do not want to consider `x||x == x` redundant, even though it is possible to prove it. After long discussions between the authors of the paper on examples like this, it became clear to us that the choice of what is redundant or not is not obvious; here there is another arbitrary choice for our algorithm to make.

Eventually we settled on the following notion of redundancy: we consider an equation redundant if it can be proved from *simpler* equations, where we measure an equation's simplicity in an ad hoc way based on the equation's length,

number of variables (which is an approximate measure of generality) etc. (For example, we will keep the idempotence law above because you cannot prove it without invoking the more complicated distributivity law.) Our justification for this choice is that a simple, general law is likely to be of interest even if it can be proved from more complex principles; in our experience, this seems to be broadly true.

Once we decide to use this "simplicity" metric most of our choices vanish and it becomes clear that our pruning algorithm should work broadly like this:

1. Sort the equations according to "simplicity".
2. Go through each equation in turn, starting with the simplest one:
   - If the equation cannot be proved from the earlier equations in the list, print it out for the user to see: it will not be pruned.
   - If it can be proved, do nothing: it has been pruned.

So now the shape of the pruning algorithm is decided. The problem we are left with is how to decide if a equation is derivable from the earlier equations.

Since derivability is undecidable, we decided to define a decidable and predictable *conservative approximation* of logical implication for equations. The approximation uses a *congruence closure* data-structure, a generalization of a union/find data-structure that maintains a congruence relation[2] over a finite subterm-closed set of terms. Like union/find, it provides two operations: unifying two congruence classes, and testing if two terms are in the same congruence class. Congruence closure is one of the key ingredients in modern SMT-solvers, and we simply reimplemented an efficient modern congruence closure algorithm following Nieuwenhuis and Oliveras [2005].

Congruence closure solves the *ground equation problem*: it can decide if a set of equations implies another equation, but only if the equations contain no variables. It goes like this: start with an empty congruence relation. For each ground equation $t = u$ that you want to assume, unify t's and u's congruence class. Then to find out if $a = b$ follows from your assumptions, simply check if a and b lie in the same congruence class. Note that the ground equation problem is decidable, and efficiently solvable: the congruence closure algorithm gives the correct answer, and very quickly.

We can try to apply the same technique to check for logical implication even when the equations are not ground. However, there is an equational inference rule for non-ground equations that is not useful for ground equations, namely that any instance of a valid equation is valid: if $t = u$ then $t\sigma = u\sigma$ (where σ is any substitution). Congruence closure does not capture this rule. Instead, we must *approximate* the rule: whenever we want to assume a non-ground equation $t = u$, instead of just unifying t's and u's congruence class, we must generate a large number of instances $t\sigma = u\sigma$ and for each of them we must unify $t\sigma$'s and $u\sigma$'s congruence class.

In other words, given a set of equations E to assume and an equation $a = b$ to try to prove, we can proceed as follows:

[2] A congruence relation is an equivalence relation that is also a congruence: if $x \equiv y$ then $C[x] \equiv C[y]$ for all contexts C.

1. Start with an empty congruence relation.
2. For each equation $t = u$ in E, generate a set of substitutions, and for each substitution σ, unify $t\sigma$'s and $u\sigma$'s congruence class.
3. Finally, to see if $a = b$ is provable from E, just check if a and b lie in the same congruence class.

This procedure is sound but incomplete: if it says $a = b$ is true, then it is; otherwise, $a = b$ is either false or just too difficult to prove.

By plugging this proof procedure into the "broad sketch" of the pruning algorithm from the previous page, we get a pruning algorithm. Doing this naively, we would end up reconstructing the congruence relation for each equation we are trying to prune; that would be wasteful. Instead we can construct the congruence relation incrementally as we go along. The final pruning algorithm looks like this, and is just the algorithm from the previous page specialised to use our above heuristic as the proof method:

1. Start with an empty congruence relation. This relation will represent all knowledge implied by the equations we have printed so far, so that if t and u lie in the same congruence class then $t = u$ is derivable from the printed equations.
2. Sort the equations according to "simplicity".
3. Go through each equation $t = u$ in turn, starting with the simplest one:
   - If t and u lie in the same congruence class, do nothing: the equation can be pruned.
   - Otherwise:
     (a) Print the equation out for the user to see: it will not be pruned.
     (b) Generate a set of substitutions; for each substitution σ, unify $t\sigma$'s and $u\sigma$'s congruence class. (This means that the congruence relation now "knows" that $t = u$ and we will be able to prune away its consequences.)

The final set of equations that is produced by the pruning algorithm is simply printed out and shown to the user as QuickSpec's output.

2.6 Which instances to generate

One thing we haven't mentioned yet is which instances of each equation $t = u$ we should generate in step 3b of the pruning algorithm—this is a crucial choice that controls the power of the algorithm. Too few instances and the pruner won't be able to prove many laws; too many and the pruner will become slow as we fill the congruence relation with hundreds of thousands of terms.

Originally we generated all instances $t\sigma = u\sigma$ such that both sides of the equation are members of the universe (recall that the universe is the large set of terms from which all equations are built). In practice this means that we would generate all instances up to a particular term depth.

This scheme allows the pruning algorithm to reason freely about the terms in the universe: we can *guarantee* to prune an equation if there is an equational proof of it where all the terms in all of the steps of the proof are in the universe, so the pruner is quite powerful and predictable.

While this scheme works pretty well for most examples, it falls down a bit when we have operators with structural properties, such as associativity. For example, generating properties about the arithmetic operator `+`, we end up with:

```
1. x+y = y+x
2. y+(x+z) = (z+y)+x
3. (x+y)+(x+z) = (z+y)+(x+x)
```

The third equation can be derived from the first two, but the proof goes through a term `x+(y+(x+z))` that lies outside of the universe, so our original scheme doesn't find the proof.

To fix this we relaxed our instance generation somewhat. According to our original scheme, when add an equation $t = u$ to our congruence relation we would generate all instances $t\sigma = u\sigma$ where both $t\sigma$ and $u\sigma$ were in the universe. Now we also generate all instances where just *one* of $t\sigma$ and $u\sigma$ is in the universe. Formally, we look at t and generate all substitutions σ such that $t\sigma$ is in the universe; then we look at u and generate all substitutions σ such that $u\sigma$ is in the universe; then we apply each of those substitutions to $t = u$ to get an instance, which we add to the congruence relation.

By relaxing the instance generation, we allow the algorithm to reason about terms that lie *outside* the universe, in a limited way. While the original scheme allows the pruner to find all equational proofs where all intermediate terms are in the universe,[3] the relaxed scheme also allows us to "jump out" of the universe for one proof step: in our proofs we are allowed to apply an equation in a way that takes us to a term *outside* the universe, provided that we follow it immediately with another proof step that takes us back into the universe. Once again, our pruner is *guaranteed* to prune a law if there is a proof of it fulfilling this restriction.

Adding the modification we just described to the algorithm, the last equation is also pruned away. The modification does not noticeably slow down QuickSpec.

2.6.1 An example

We now demonstrate our pruning algorithm on the running booleans example. To make it more interesting we add another term to the universe, `false&&(x&&false)`. Our pruning algorithm sorts the initial set of equations by simplicity, giving the following:

```
1. x&&x == x
2. y&&y == y
```

[3] Although the pruning algorithm does not exactly search for *proofs*, it is useful to characterise the pruning algorithm's power by what proofs it can find: "if it is possible to prove the equation using a proof of *this form* then the equation will be pruned"

```
3. x&&y == y&&x
4. x&&false == false
5. y&&false == false
6. false&&x == false
7. false&&y == false
8. false&&false == false
9. false&&(x&&false) == false.
```

We start with an empty congruence relation ≡ in which every term is in its own congruence class:

```
{x} {y} {false} {x&&x} {y&&y} {x&&y} {y&&x}
{x&&false} {y&&false} {false&&x} {false&&y}
{false&&false} {false&&(x&&false)}
```

Starting from equation 1, we see that `x&&x` and `x` are in different congruence classes (we can't prove them equal) so we print out `x&&x == x` as an equation. We add its instances to the congruence-closure by unifying `x&&x` with `x`, `y&&y` with `y` and `false&&false` with `false`.[4] The congruence relation now looks like this:

```
{x, x&&x} {y, y&&y} {false, false&&false} {x&&y} {y&&x}
{x&&false} {y&&false} {false&&x} {false&&y}
{false&&(x&&false)}
```

Coming to equation 2, we see that `y&&y` and `y` *are* in the same congruence class, so we prune away the equation: we can prove it from the previous equation. Equation 3 isn't provable, so we print it out and add some instances to the congruence closure data structure—`x&&y` ≡ `y&&x`, `x&&false` ≡ `false&&x`, `y&&false` ≡ `false&&y`. Now the congruence relation is as follows:

```
{x, x&&x} {y, y&&y}, {false, false&&false} {x&&y, y&&x}
{x&&false, false&&x} {y&&false, false&&y}
{false&&(x&&false)}
```

Equation 4—`x&&false == false` isn't provable either, so we print it out. We generate some instances as usual—`x&&false` ≡ `false`, `y&&false` ≡ `false`, `false&&false` ≡ `false`. This results in the following congruence relation:

```
{x, x&&x} {y, y&&y}
{false, x&&false, false&&x, y&&false, false&&y,
 false&&false, false&&(x&&false)}
{x&&y, y&&x}
```

Notice that `false&&(x&&false)` is now in the same congruence class as `false`, even though we never unified it with anything. This is the extra feature that congruence closure provides over union/find—since we told it that `x&&false` ≡ `false` and `false&&false` ≡ `false`, it deduces by itself that `false&&(x&&false)` ≡ `false&&false` ≡ `false`.

[4] According to our relaxed instance-generation scheme from above, we should generate even more instances because `x` can range over any term in the universe, but we ignore this for the present example.

Looking at the remainder of our equations, both sides are always in the same congruence class. So all the remaining equations are pruned away and the final set of equations produced by QuickSpec is

```
1. x&&x == x
2. x&&y == y&&x
3. x&&false == false
```

as we desired.

2.7 Alternative pruning methods that Don't Work

The above pruning algorithm may seem overly sophisticated (although it is rather short and sweet in practice because the congruence closure algorithm does all the hard work). Wouldn't a simpler algorithm be enough? We claim that the answer is no: in this section we present two such simpler pruning algorithms, which we used in earlier versions of QuickSpec; both algorithms fail to remove many real-life redundant laws.

We present these failed algorithms in the hope that they illustrate how well-behaved our current pruning algorithm is.

2.7.1 Instance-based pruning

In our first preliminary experiments with QuickSpec, we used the following pruning algorithm: simply delete an equation if it's an instance of another one.

For our boolean example this leads to the following set of laws:

```
1. x&&x == x
2. x&&y == y&&x
3. x&&false == false
4. false&&x == false
```

which is not satisfactory: laws 3 and 4 state the same thing modulo commutativity. However, to prove law 4 you need to use laws 3 and 2 together, and this simplistic pruning method is not able to do that: it will only prune a law if it's an instance of *one* other law. In all real examples this simplistic algorithm is hopelessly inadequate.

2.7.2 Rewriting-based pruning

We next observed that we ought to be able to delete the equation `t == u` (`u` being `t`'s representative), if we can use the other equations as *rewrite rules* to simplify `t` to `u`. (For this to work, we must have a total order defining the *simplicity* of a term, we may only rewrite each term to a simpler term, and both sides of any rewrite rule we use must be simpler than the thing we are trying to rewrite.)

It turns out not to be necessary to simplify `t` all the way to `u`: if we can rewrite `t` to a simpler term `t'`, we may unconditionally delete the equation `t == u`.

The fact that we only need to consider *a single rewrite step* was crucial to the efficiency of this algorithm.

(The reason why this one-step reduction works is that we know that `t`, `t'` and `u` must all live in the same equivalence class. Therefore our initial set of equations will contain `t' == u`. If the pruner is sound we can prove `t' == u` from QuickSpec's output, and since our rewrite rule proves `t == t'` we can also prove `t == u` and therefore may prune it away.)

An aside: it is tempting to remove the restriction that `t'` must be smaller than `t`, but this leads to circularity: we could prune `t == u` by rewriting `t` to `t'` and then `t' == u` by rewriting `t'` to `t`.

This algorithm was able to prove any law that could be proved by repeatedly simplifying the left-hand side until you got to the right-hand side—much better than the instance-based pruning. Note also that we never need to combine rewrite steps—we only try each possible individual rewrite step against each equation—so it's fairly efficient when implemented.

Let's see it in action on our booleans example. The equations are first sorted in order of simplicity:

```
1. x&&x == x
2. y&&y == y
3. x&&y == y&&x
4. x&&false == false
5. y&&false == false
6. false&&x == false
7. false&&y == false
8. false&&false == false
```

Equation 1 is not provable and is printed out.

Taking equation 2, we can rewrite the left-hand side, `y&&y`, to `y` using equation 1. This leaves us with `y == y`, which is simpler than equation 2, so we delete equation 2. (So rewriting-based pruning subsumes instance-based pruning.)

Equations 3 and 4 are not provable. Equation 5 is pruned in a similar way to equation 2.

Equation 6 is the equation that the instance-based pruning algorithm couldn't remove. Rewriting-based pruning has no problem here: we apply equation 3 to the left-hand side to rewrite `false&&x == x` to `x&&false == x`, which is simpler according to our order (it's equation 4, and the equations are sorted by simplicity), so the pruner deletes equation 6.

Equations 7 and 8 get pruned just like equations 2 and 5.

Why it doesn't work So why don't we use this pruning method any more? It seems promising at first glance and is quite easy to implement. Indeed, we used it in QuickSpec for quite a long time, and it works well on some kinds of examples, such as lists. However, it is unable to apply a rewrite step "backwards", starting from a simpler term to get to a more complicated term; this is unfortunately often necessary. Problematic cases for this algorithm were:

1. Commutativity laws such as `x&&y == y&&x` have no natural orientation.

So this algorithm may be allowed to rewrite `t&&u` to `u&&t` or it may not; this is determined almost "randomly" depending on which term our equation order considers simpler. If the proof of an equation uses commutativity in the "wrong" direction, the rewriting-based pruner won't be able to delete the equation.

2. If we have in our signature one operator that distributes over another, for example `x&&(y||z) == (x&&y)||(x&&z)`, the pruner will only be able to apply this law from right to left (contracting a term). However, there are many laws that we can only prove by using distributivity to expand a term followed by simplifying it. The rewriting-based pruner will not be able to filter out those laws.

We tried to fix these flaws by allowing the pruner to use multiple rewrite steps when simplifying `t == u`: after the last step we must have an equation simpler than `t == u`, but the intermediate equations can be anything. Then we could for example have a proof that starts by applying a distributivity law and then simplifying. However, naively searching for rewrite proofs of long lengths leads instantly to exponential blowup; to avoid that we would have needed lots of machinery from term rewriting (e.g. using unfailing Knuth-Bendix completion to get a confluent rewrite system), which would have greatly complicated the pruning algorithm. The "improved" algorithm would have been both more complicated and less effective than the congruence-closure-based algorithm we use now.

Rewriting-based pruning explodes How big of a problem was this in practice? To answer, we list just a small selection of the laws produced by the rewriting-based pruning algorithm when we ran QuickSpec on a sets example. The signature contains four operations, `new` which returns the empty set, `add_element` which inserts one element into a set, and the usual set operators `union` and `intersection`. The rewriting-based pruning algorithm printed out these laws, along with dozens of others:

```
union(S,S) == S
union(S,new()) == S
union(T,S) == union(S,T)
union(U,union(S,T)) == union(S,union(T,U))
union(S,union(S,T)) == union(S,T)
union(union(S,T),union(T,U)) == union(S,union(T,U))
union(union(S,U),union(T,V)) == union(union(S,T),union(U,V))
add_element(X,add_element(X,S)) == add_element(X,S)
add_element(Y,add_element(X,S)) ==
  add_element(X,add_element(Y,S))
union(S,add_element(X,T)) == add_element(X,union(S,T))
union(add_element(X,S),add_element(X,T)) ==
  add_element(X,union(S,T))
union(add_element(X,S),add_element(Y,S)) ==
  add_element(X,add_element(Y,S))
union(add_element(X,S),union(S,T)) ==
```

```
  add_element(X,union(S,T))
union(add_element(X,T),add_element(Y,S)) ==
  union(add_element(X,S),add_element(Y,T))
union(add_element(X,T),union(S,U)) ==
  union(add_element(X,S),union(T,U))
intersection(intersection(S,T),union(S,U)) ==
  intersection(S,T)
intersection(intersection(S,T),union(T,U)) ==
  intersection(S,T)
intersection(intersection(S,U),intersection(T,V)) ==
  intersection(intersection(S,T),intersection(U,V))
intersection(intersection(S,U),union(T,U)) ==
  intersection(S,U)
intersection(intersection(T,U),union(S,T)) ==
  intersection(T,U)
```

Terrible! Notice how many of the equations are just simple variations of each other. By contrast, the current implementation of QuickSpec, using the pruning algorithm described in section 2.5, returns these 17 laws only:

```
1.  intersection(T,S) == intersection(S,T)
2.  union(T,S) == union(S,T)
3.  intersection(S,S) == S
4.  intersection(S,new()) == new()
5.  union(S,S) == S
6.  union(S,new()) == S
7.  add_element(Y,add_element(X,S)) ==
      add_element(X,add_element(Y,S))
8.  intersection(T,intersection(S,U)) ==
      intersection(S,intersection(T,U))
9.  union(S,add_element(X,T)) == add_element(X,union(S,T))
10. union(T,union(S,U)) == union(S,union(T,U))
11. intersection(S,add_element(X,S)) == S
12. intersection(S,union(S,T)) == S
13. union(S,intersection(S,T)) == S
14. intersection(add_element(X,S),add_element(X,T)) ==
      add_element(X,intersection(S,T))
15. intersection(union(S,T),union(S,U)) ==
      union(S,intersection(T,U))
16. union(add_element(X,S),add_element(X,T)) ==
      add_element(X,union(S,T))
17. union(intersection(S,T),intersection(S,U)) ==
      intersection(S,union(T,U))
```

This is still not ideal—there is some repetition between `add_element` and `union` (more on that in section 2.8), but is *much* better.

Comparison with the current algorithm In our experience, pruning by rewriting is too *brittle* and *unpredictable*: it depends on ordering the terms in

exactly the right way so that all the laws we want to erase can be proved purely by simplification—we can never rewrite a smaller term to a larger term. It is far too sensitive to the exact term order we use (to get good results we had to use a carefully-tuned, delicate heuristic), and blows up when there is no natural term order to use, as is the case when there are commutative operators, or when a proof needs to expand a term before simplifying it, as is the case with distributive operators. Attempts to patch this up by searching for "expansion" steps as well as simplification steps in proofs complicate the algorithm to an unreal extent, for example requiring the use of Knuth-Bendix completion to get a useful algorithm.

By contrast, our current algorithm that uses congruence closure is predictable and well-behaved:

- It is strictly more powerful than the rewriting-based pruner.
- It does no search or rewriting, instead merely recording a set of facts, so it has no trouble dealing with commutativity or distributivity or any other strange algebraic properties.
- It is much less sensitive to changes in the equation order than the pruner that used rewriting: the equation order affects which laws you may use as assumptions when proving an equation, but not how those laws may be applied as in the rewriting-based pruner.
- On most examples it works well; perhaps more importantly, we know of no examples where it does *really badly*, which is certainly not the case with the pruner it replaced. We are able to just rely on it to do the right thing whatever program we give QuickSpec, which is the most important thing.

2.8 Definitions

Recall our sets example from section 2.7.2. We have four operators, `new` that returns the empty set, `add_element` that adds an element to a set, and `union` and `intersection`. Running QuickSpec on this signature, we get slightly unsatisfactory results:

```
1.  intersection(T,S) == intersection(S,T)
2.  union(T,S) == union(S,T)
3.  intersection(S,S) == S
4.  intersection(S,new()) == new()
5.  union(S,S) == S
6.  union(S,new()) == S
7.  add_element(Y,add_element(X,S)) ==
      add_element(X,add_element(Y,S))
8.  intersection(T,intersection(S,U)) ==
      intersection(S,intersection(T,U))
9.  union(S,add_element(X,T)) == add_element(X,union(S,T))
10. union(T,union(S,U)) == union(S,union(T,U))
11. intersection(S,add_element(X,S)) == S
```

```
12. intersection(S,union(S,T)) == S
13. union(S,intersection(S,T)) == S
14. intersection(add_element(X,S),add_element(X,T)) ==
      add_element(X,intersection(S,T))
15. intersection(union(S,T),union(S,U)) ==
      union(S,intersection(T,U))
16. union(add_element(X,S),add_element(X,T)) ==
      add_element(X,union(S,T))
17. union(intersection(S,T),intersection(S,U)) ==
      intersection(S,union(T,U))
```

These results state everything we would like to know about union and intersection, but there are perhaps more laws than we would like to see. Several laws appear in two variants, one for `union` and one for `add_element`.

This suggests that `union` and `add_element` are similar somehow. And indeed they are: `add_element` is the special case of `union` where one set is a singleton set ($S \cup \{x\}$). The way to reduce the number of equations is to replace `add_element` by a function `unit` that returns a singleton set. This function is simpler, so we expect better laws, but the API remains as expressive as before because `add_element` can be defined using `unit` and `union`. If we do that we get fewer laws:

```
1.  intersection(T,S) == intersection(S,T)
2.  union(T,S) == union(S,T)
3.  intersection(S,S) == S
4.  intersection(S,new()) == new()
5.  union(S,S) == S
6.  union(S,new()) == S
7.  intersection(T,intersection(S,U)) ==
      intersection(S,intersection(T,U))
8.  union(T,union(S,U)) == union(S,union(T,U))
9.  intersection(S,union(S,T)) == S
10. union(S,intersection(S,T)) == S
11. intersection(union(S,T),union(S,U)) ==
      union(S,intersection(T,U))
12. union(intersection(S,T),intersection(S,U)) ==
      intersection(S,union(T,U))
```

Now all the laws are recognisable as standard set theory ones, so we should conclude that there is not much redundancy here. Much better!

This technique is more widely applicable: whenever we have a redundant operator we will often get better results from QuickSpec if we remove it from the signature. QuickSpec in fact alerts us that an operator is redundant by printing out a *definition* of that operator in terms of other operators. In our case, when we ran QuickSpec the first time it also printed the following:

```
add_element(X,S) := union(S,add_element(X,new()))
```

In other words, `add_element(X,S)` is the union of `S` and the singleton set `{X}`, which we can construct with `add_element(X,new())`.

What QuickSpec looks for when it searches for definitions is a pair of equal terms in the equivalence relation satisfying the following conditions:

- One term must be a function call with all arguments distinct variables. In our case, this is `add_element(X,S)`. This is the left-hand side of the definition.
- The definition should not be circular; for example, we should not emit `union(S,T) := union(T,S)` as a definition. One possibility would be to forbid the right-hand side of a definition from referring to the function we're trying to define. However, this is too restrictive: in the definition of `add_element`, we use `add_element` on the right-hand side but we use a *special case* of `add_element` to construct a singleton set. We capture this by allowing the right-hand side of the definition to call the function it defines, but with one restriction: there must be a variable on the left-hand side of the definition that does not appear in the "recursive" call. In our case, the left-hand side mentions the variable `S` and the recursive call to `add_element` does not, so we conclude that the recursive call is a special case of `add_element` rather than a circular definition.

2.9 The depth optimisation

QuickSpec includes one optimisation to reduce the number of terms generated. We will first motivate the optimisation and then explain it in more detail.

Suppose we have run QuickSpec on an API of boolean operators with a depth limit of 2, giving (among others) the law `x&&x==x`. But now, suppose we want to increase the depth limit on terms from 2 to 3. Using the algorithm described above, we would first generate all terms of depth 3, including such ones as `x&&y` and `(x&&x)&&y`. But these two terms are obviously equivalent (since we know that `x&&x==x`), we won't get any more laws by generating both of them, and we ought to generate only `x&&y` and not `(x&&x)&&y`.

The observation we make is that, if two terms are equal (like `x&&x` and `x` above), we ought to pick one of them as the "canonical form" of that expression; we avoid generating any term that has a non-canonical form as a subterm. In this example, we don't generate `(x&&x)&&y`, because it has `x&&x` as a subterm. (We do still generate `x&&x` on its own, otherwise we wouldn't get the law `x&&x==x`.)

The depth optimisation applies this observation, and works quite straightforwardly. If we want to generate all terms up to depth 3, say, we first generate all terms up to depth 2 and sort them into equivalence classes by testing. The representatives of those classes we intend to be the "canonical forms" we mentioned above. Then, for terms of depth 3, we generate only those terms for which all the direct subterms are the representatives of their equivalence class. In the example above, we have an equivalence class `{x, x&&x}`; `x` is the representative. So we will generate terms that contain `x` as a direct subterm but not ones that contain `x&&x`.

We can justify why this optimisation is sound. If we choose not to generate a

term `t` with canonical form `t'`, and if testing would have revealed an equation `t==u`, we will also generate an equation `t'==u`.[5] We also will have generated laws that imply `t==t'` since each direct subterm of `t` is equal to the corresponding subterm of `t'` (in the booleans example the law in question would be `x&&x == x`), and therefore `t==u` is redundant. (It is also the case that our pruner would've filtered out `t==u`.)

This optimisation makes a very noticeable difference to the number of terms generated. For a large list signature, the number of terms goes down from 21266 to 7079. For booleans there is a much bigger difference, since so many terms are equal: without the depth optimisation we generate 7395 terms, and with it 449 terms. Time-wise, the method becomes an order of magnitude faster.

2.10 Generating Test Data

As always with random testing tools, the quality of the test data determines the quality of the generated equations. As such, it is important to provide good test data generators, that fit the program at hand. In our property-based random testing tool QuickCheck [Claessen and Hughes, 2000], we have a range of test data generators for standard types, and a library of functions for building custom generators. QuickSpec simply reuses QuickCheck's random data generators.

As an example of what can happen if we use an inappropriate generator, consider generating laws for an API including the following functions:

```
isPrefixOf :: [Elem] -> [Elem] -> Bool
null       :: [Elem] -> Bool
```

If one is not careful in defining the list generator, QuickSpec might end up producing the law `isPrefixOf xs ys == null xs`. Why? Because for two randomly generated lists, it is very unlikely that one is a prefix of the other, unless the first is empty. So, there is a risk that the interesting test cases that separate `isPrefixOf xs ys` and `null xs` will not be generated. The problem can be solved by making sure that the generator used for random lists is likely to pick the list elements from a small domain. Thus, just as in using QuickCheck, creating custom generators is sometimes necessary to get correct results.

We have to point out that this does not happen often even if we are careless about test data generation: as noted earlier, in our implementation we keep on refining the equivalence relation until it has been stable for 200 iterations, which gives QuickSpec a better chance of falsifying hard-to-falsify equations.

3 Case Studies

In this section, we present two case studies using QuickSpec. Our goal is primarily to derive *understanding* of the code we test. In many cases, the

[5] This relies on `t'` not having greater depth than `t`, which requires the term ordering to always pick the representative of an equivalence class as a term with the smallest depth.

specifications generated by QuickSpec are initially disappointing—but by *extending the signature with new operations* we are able to arrive at concise and perspicuous specifications. Arguably, selecting the right operations to specify is a key step in formulating a good specification, and one way to see QuickSpec is as a tool to support exploration of this design space.

3.1 Case Study #1: Leftist Heaps in Haskell

A *leftist heap* [Okasaki, 1998] is a data structure that implements a priority queue. A leftist heap provides the usual heap operations:

```
empty :: Heap
isEmpty :: Heap -> Bool
insert :: Elem -> Heap -> Heap
findMin :: Heap -> Elem
deleteMin :: Heap -> Heap
```

When we tested this signature with the variables `h, h1, h2 :: Heap` and `x, y, z :: Elem`. then QuickSpec generated a rather incomplete specification. The specification describes the behaviour of `findMin` and `deleteMin` on empty and singleton heaps:

```
findMin empty == undefined
findMin (insert x empty) == x
deleteMin empty == undefined
deleteMin (insert x empty) == empty
```

It shows that the order of insertion into a heap is irrelevant:

```
insert y (insert x h) == insert x (insert y h),
```

Apart from that, it only contains the following equation:

```
isEmpty (insert x h1) == isEmpty (insert x h)
```

This last equation is quite revealing—obviously, we would expect both sides to be `False`, which explains why they are equal. But why doesn't QuickSpec just print the equation `isEmpty (insert x h) == False`? The reason is that `False` is not in our signature! When we add it to the signature, then we do indeed obtain the simpler form instead of the original equation above.[6]

In general, when a term is found to be equal to a renaming of itself with different variables, then this is an indication that a constant should be added to the signature, and in fact QuickSpec prints a suggestion to do so.

Generalising a bit, since `isEmpty` returns a `Bool`, it's certainly sensible to give QuickSpec operations that manipulate booleans. We added the remaining boolean connectives `True`, `&&`, `||` and `not`; one new law appeared that we couldn't express before, `isEmpty empty == True`.

[6] For completeness, we will list all of the new laws that QuickSpec produces every time we change the signature.

3.1.1 Merge

Leftist heaps actually provide one more operation than those we encountered so far: merging two heaps.

```
merge :: Heap -> Heap -> Heap
```

If we run QuickSpec on the new signature, we get the fact that `merge` is commutative and associative and has `empty` as a unit element:

```
merge h1 h == merge h h1
merge h1 (merge h h2) == merge h (merge h1 h2)
merge h empty == h
```

We get nice laws about `merge`'s relationship with the other operators:

```
merge h (insert x h1) == insert x (merge h h1)
isEmpty h && isEmpty h1 == isEmpty (merge h h1)
```

We also get some curious laws about merging a heap with itself:

```
findMin (merge h h) == findMin h
merge h (deleteMin h) == deleteMin (merge h h)
```

These are *all* the equations that are printed. Note that there are no redundant laws here. As mentioned earlier, our testing method guarantees that this set of laws is *complete*, in the sense that any valid equation over our signature, which is not excluded by the depth limit, follows from these laws.

3.1.2 With Lists

We can get useful laws about heaps by relating them to a more common data structure, *lists*. First, we need to extend the signature with operations that convert between heaps and lists:

```
fromList :: [Elem] -> Heap
toList :: Heap -> [Elem]
```

`fromList` turns a list into a heap by folding over it with the `insert` function; `toList` does the reverse, deconstructing a heap using `findMin` and `deleteMin`. We should also add a few list operations mentioned earlier:

```
(++) :: [Elem] -> [Elem] -> [Elem]
tail :: [Elem] -> [Elem]
(:) :: Elem -> [Elem] -> [Elem]
[] :: [Elem]
sort :: [Elem] -> [Elem]
```

and variables `xs, ys, zs :: [Elem]`. Now, QuickSpec discovers many new laws. The most striking one is

```
toList (fromList xs) == sort xs.
```

This is the definition of heapsort! The other laws indicate that our definitions of `toList` and `fromList` are sensible:

```
sort (toList h) == toList h
fromList (toList h) == h
fromList (sort xs) == fromList xs
fromList (ys++xs) == fromList (xs++ys)
```

The first law says that `toList` produces a sorted list, and the second one says that `fromList . toList` is the identity (up to `==` on heaps, which actually applies `toList` to each operand and compares them!). The other two laws suggest that the order of `fromList`'s input doesn't matter.

We get a definition by pattern-matching of `fromList` (read the second and third equations from right to left):

```
fromList [] == empty
insert x (fromList xs) == fromList (x:xs)
merge (fromList xs) (fromList ys) == fromList (xs++ys)
```

We also get a family of laws relating heap operations to list operations:

```
toList empty == []
head (toList h) == findMin h
toList (deleteMin h) == tail (toList h)
```

We can think of `toList h` as an abstract model of `h`—all we need to know about a heap is the sorted list of elements, in order to predict the result of any operation on that heap. The heap itself is just a clever representation of that sorted list of elements.

The three laws above define `empty`, `findMin` and `deleteMin` by how they act on the sorted list of elements—the model of the heap. For example, the third law says that applying `deleteMin` to a heap corresponds to taking the `tail` in the abstract model (a sorted list). Since `tail` is obviously the correct way to remove the minimum element from a sorted list, this equation says exactly that `deleteMin` is correct![7]

So these three equations are a *complete* specification of the three functions `empty`, `findMin` and `deleteMin`!

If we want to extend this to a complete specification of heaps, we must add operators to insert an element into a sorted list, to merge two sorted lists, and to test if a sorted list is empty...

```
insertL :: Elem -> [Elem] -> [Elem]
mergeL :: [Elem] -> [Elem] -> [Elem]
null :: [Elem] -> Bool
```

...and our reward is three laws asserting that the functions `insert`, `merge` and `isEmpty` are correct:

[7] This style of specification is not new and goes back to Hoare [1972].

```
toList (insert x h) == insertL x (toList h)
mergeL (toList h) (toList h1) == toList (merge h h1)
null (toList h) == isEmpty h
```

We also get another law about `fromList` to go with our earlier collection. This one says essentially that `mergeL xs ys` contains each of the members of both `xs` and `ys` exactly once:

```
fromList (mergeL xs ys) == fromList (xs++ys)
```

This section highlights the importance of choosing a rich set of operators when using QuickSpec. There are often useful laws about a library that mention functions from unrelated libraries; the more such functions we include, the more laws QuickSpec can find. In the end, we got a complete specification of heaps (and heapsort, as a bonus!) by including list functions in our testing.

It's not always obvious *which* functions to add to get better laws. In this case, there were several reasons for choosing lists: they're well-understood, there are operators that convert heaps to and from lists, and sorted lists form a model of priority queues.

3.1.3 Buggy Code

What happens when the code under test has a bug? To find out, we introduced a fault into `toList`. The buggy version of `toList` doesn't produce a sorted list, but rather the elements of the heap in an arbitrary order.

We were hoping that some laws would fail, and that QuickSpec would produce *specific instances* of some of those laws instead. This happened: whereas before, we had many useful laws about `toList`, afterwards, we had only two:

```
toList empty == []
toList (insert x empty) == x:[]
```

Two things stand out here: first, the law `sort (toList h) == toList h` does not appear, so we know that the buggy `toList` doesn't produce a sorted result. Second, we *only get equations about empty and singleton heaps*, not about heaps of arbitrary size. QuickSpec is unable to find *any* specification of `toList` on nontrivial heaps, which suggests that the buggy `toList` *has* no simple specification.

3.1.4 A trick

We finish with a "party trick": getting QuickSpec to discover how to implement `insert` and `deleteMin`. We hope to run QuickSpec and see it print equations of the form `insert x h = ?` and `deleteMin h = ?`.

We need to prepare the trick first; if we just run QuickSpec straight away, we won't get either equation. There are two reasons, each of which explains the disappearance of one equation.

First, it's impossible to implement `deleteMin` using only the leftist heap API, so there's no equation for QuickSpec to print. To give QuickSpec a chance, we need to reveal the representation of leftist heaps; they're really binary trees. So we add the functions

```
leftBranch :: Heap -> Heap
rightBranch :: Heap -> Heap
```

to the signature. Of course, no implementation of leftist heaps would export these functions, this is only for the trick.

Secondly, QuickSpec won't bother to print out the definition of `insert`: it's easily derivable from the other laws, so QuickSpec considers it boring. Actually, in most ways, it *is* pretty boring; the one thing that makes it interesting is that it defines `insert`, but QuickSpec takes no notice of that.

Fortunately, we have a card up our sleeve: QuickSpec prints a list of *definitions*, equations that define an operator in terms of other operators, as we saw in section 2.8. The real purpose of this is to suggest redundant operators, but we will use it to see the definition of `insert` instead.

Finally, we also need to be careful—previously, we were treating our heap as an *abstract data type*, so that two heaps would be equal if they had the same elements. But `leftBranch` and `rightBranch` peek into the internals of the heap, so they can distinguish heaps that are morally the same. So we had better tell QuickSpec that equality should check the representation of the heap and not its contents.

Everything in place at last, we run QuickSpec. And—hey presto!—out come the equations

```
insert x h = merge h (insert x empty)
deleteMin h = merge (leftBranch h) (rightBranch h)
```

That is, you can insert an element by merging with a unit heap that just contains that element, or delete the minimum element—which happens to be stored at the root of the tree—by merging the root's branches.

3.2 Case Study #2: Understanding a Fixed Point Arithmetic Library in Erlang

We used QuickSpec to try to understand a library for fixed point arithmetic, developed by a South African company, which we were previously unfamiliar with. The library exports 16 functions, which is rather overwhelming to analyze in one go, so we decided to generate equations for a number of different subsets of the API instead. In this section, we give a detailed account of our experiments and developing understanding.

Before we could begin to use QuickSpec, we needed a QuickCheck generator for fixed point data. We chose to use one of the library functions to ensure a valid result, choosing one which seemed able to return arbitrary fixed point values:

```
fp() -> ?LET({N,D},{largeint(),nat()},from_minor_int(N,D)).
```

That is, we call `from_minor_int` with random arguments. We suspected that `D` is the precision of the result—a suspicion that proved to be correct.

3.2.1 Addition and Subtraction

We began by testing the `add` operation, deriving commutativity and associativity laws as expected. Expecting laws involving zero, we defined

```
zero() -> from_int(0)
```

and added it to the signature, obtaining as our reward a unit law,

```
add(A,zero()) == A.
```

The next step was to add subtraction to the signature. However, this led to several very similar laws being generated—for example,

```
add(B,add(A,C)) == add(A,add(B,C))
add(B,sub(A,C)) == add(A,sub(B,C))
sub(A,sub(B,C)) == add(A,sub(C,B))
sub(sub(A,B),C) == sub(A,add(B,C))
```

To relieve the problem, we added another derived operator to the signature instead:

```
negate(A) -> sub(zero(),A).
```

and observed that the earlier family of similar laws was no longer generated, replaced by a single one, `add(A,negate(B)) == sub(A,B)`. Thus by *adding* a new auxiliary function to the signature, `negate`, we were able to *reduce* the complexity of the specification considerably.

After this new equation was generated by QuickSpec, we tested it extensively using *QuickCheck*. Once confident that it held, we could safely *replace* `sub` in our signature by `add` and `negate`, without losing any other equations. Once we did this, we obtained a more useful set of new equations:

```
add(negate(A),add(A,A)) == A
add(negate(A),negate(B)) == negate(add(A,B))
negate(negate(A)) == A
negate(zero()) == zero()
```

These are all very plausible—what is striking is the *absence* of the following equation:

```
add(A,negate(A)) == zero()
```

When an expected equation like this is missing, it's easy to formulate it as a QuickCheck property and find a counterexample, in this case `{fp,1,0,0}`. We discovered by experiment that `negate({fp,1,0,0})` is actually the same value! This strongly suggests that this is an alternative representation of zero (`zero()` evaluates to `{fp,0,0,0}` instead).

3.2.2 $0 \neq 0$

It is reasonable that a fixed point arithmetic library should have different representations for zero of different precisions, but we had not anticipated this. Moreover, since we want to derive equations involving zero, the question arises of *which zero* we would like our equations to contain! Taking our cue from the missing equation, we introduced a new operator `zero_like(A) -> sub(A,A)` and then derived not only `add(A,negate(A)) == zero_like(A)` but a variety of other interesting laws. These two equations suggest that the result of `zero_like` depends only on the number of decimals in its argument,

```
zero_like(from_int(I)) == zero()
zero_like(from_minor_int(J,M)) ==
  zero_like(from_minor_int(I,M))
```

this equation suggests that the result has the *same* number of decimals as the argument,

```
zero_like(zero_like(A)) == zero_like(A)
```

while these two suggest that the number of decimals is preserved by arithmetic.

```
zero_like(add(A,A)) == zero_like(A)
zero_like(negate(A)) == zero_like(A)
```

It is not in general true that `add(A,zero_like(B)) == A` which is not so surprising—the precision of `B` affects the precision of the result. QuickSpec does find a more restricted property, `add(A,zero_like(A)) == A`.

The following equations suggest that the precision of the results of `add` and `negate` depend only on the precision of the arguments, not their values:

```
add(zero_like(A),zero_like(B)) == zero_like(add(A,B))
negate(zero_like(A)) == zero_like(A)
```

3.2.3 Multiplication and Division

When we added multiplication and division operators to the signature, then we followed a similar path, and were led to introduce `reciprocal` and `one_like` functions, for similar reasons to `negate` and `zero_like` above. One interesting equation we discovered was this one:

```
divide(one_like(A),reciprocal(A)) ==
  reciprocal(reciprocal(A))
```

The equation is clearly true, but why does it say `reciprocal(reciprocal(A))` instead of just `A`? The reason is that the left-hand side raises an exception if `A` is zero, and so the right-hand side must do so also—which `reciprocal(reciprocal(A))` does.

We obtain many equations that express things about the precision of results, such as

```
multiply(B,zero_like(A)) == zero_like(multiply(A,B))
multiply(from_minor_int(I,N),from_minor_int(J,M)) ==
           multiply(from_minor_int(I,M),from_minor_int(J,N))
```

where the former expresses the fact that the precision of the zero produced depends both on `A` and `B`, and the latter expresses

$$i \times 10^{-m} \times j \times 10^{-n} = i \times 10^{-n} \times j \times 10^{-m}$$

That is, it is in a sense the commutativity of multiplication in disguise.

One equation we expected, but did *not* see, was the distributivity of multiplication over addition. Alerted by its absence, we formulated a corresponding QuickCheck property,

```
prop_multiply_distributes_over_add() ->
    ?FORALL({A,B,C},{fp(),fp(),fp()},
        multiply(A,add(B,C)) ==
          add(multiply(A,B),multiply(A,C))).
```

and used it to find a counterexample:

```
A = {fp,1,0,4}, B = {fp,1,0,2}, C = {fp,1,1,4}
```

We used the library's `format` function to convert these to strings, and found thus that $A = 0.4, B = 0.2, C = 1.4$. Working through the example, we found that multiplying `A` and `B` returns a representation of 0.1, and so we were alerted to the fact that `multiply` rounds its result to the precision of its arguments.

3.2.4 Understanding Precision

At this point, we decided that we needed to understand how the precision of results was determined, so we defined a function `precision` to extract the first component of an `{fp,...}` structure, where we suspected the precision was stored. We introduced a `max` function on naturals, guessing that it might be relevant, and (after observing the term `precision(zero())` in generated equations) the constant natural zero. QuickSpec then generated equations that tell us rather precisely how the precision is determined, including the following:

```
max(precision(A),precision(B)) == precision(add(A,B))
precision(divide(zero(),A)) == precision(one_like(A))
precision(from_int(I)) == 0
precision(from_minor_int(I,M)) == M
precision(multiply(A,B)) == precision(add(A,B))
precision(reciprocal(A)) == precision(one_like(A))
```

The first equation tells us the addition uses the precision of whichever argument has the most precision, and the fifth equation tells us that multiplication does the same. The second and third equations confirm that we have understood the representation of precision correctly. The second and sixth equations reveal that our definition of `one_like(A)` raises an exception when `A` is zero—this is why we do not see

```
precision(one_like(A)) == precision(A).
```

The second equation is more specific than we might expect, and in fact it is true that

```
precision(divide(A,B)) ==
  max(precision(A),precision(one_like(B)))
```

but the right-hand side exceeds our depth limit, so QuickSpec cannot discover it.

If we could discover conditional equations, then QuickSpec might discover instead that

```
B/=zero_like(B) ==>
  precision(divide(A,B)) == precision(add(A,B))
```

a property which we verified with QuickCheck.

3.3 Adjusting Precision

The library contained two operations whose meaning we could not really guess from their names, `adjust` and `shr`. Adding `adjust` to the signature generated a set of equations including the following:

```
adjust(A,precision(A)) == A
precision(adjust(A,M)) == M
zero_like(adjust(A,M)) == adjust(zero(),M)
adjust(zero_like(A),M) == adjust(zero(),M)
```

These equations make it fairly clear that `adjust` sets the precision of its argument. We also generated an equation relating double to single adjustment:

```
adjust(adjust(A,M),0) == adjust(A,0)
```

We generalised this to

```
N =< M ==> adjust(adjust(A,M),N) == adjust(A,N),
```

a law which QuickSpec might well have generated if it could produce conditional equations. We tested the new equation with QuickCheck, and discovered it to be false! The counterexample QuickCheck found shows that the problem is caused by rounding: adjusting 0.1045 to three decimal places yields 0.105, and adjusting this to two decimals produces 0.11. Adjusting the original number to two decimals in one step produces 0.10, however, which is different. In fact, the original equation that QuickSpec found above is also false—but several hundred tests are usually required to find a counterexample. This shows the importance of testing the most interesting equations that QuickSpec finds more extensively—occasionally, it does report falsehoods. Had we written a test data generator that tried to provoke interesting rounding behaviours we mightn't have encountered these false equations.

3.4 `shr`: Problems with Partiality

Adding `shr` to the signature, too, we at first obtained several rather complex equations, of which this is a typical example:

```
adjust(shr(zero(),I),precision(A)) == shr(zero_like(A),I)
```

All the equations had one thing in common: `shr` appeared on both sides of the equation, with the same second argument. Eventually we realised why: `shr` is a *partial* function, which raises an exception when its second argument is negative—so QuickSpec produced equations with `shr` on both sides so that exceptions would be raised in the same cases. We changed the signature to declare `shr`'s second argument to be `nat()` rather than `int()`, whereupon QuickSpec produced simple equations as usual.

QuickSpec told us how the precision of the result is determined:

```
precision(shr(A,M)) == precision(A)
```

Other informative equations were

```
shr(shr(A,N),M) == shr(shr(A,M),N)
shr(A,0) == A
shr(zero_like(A),M) == zero_like(A)
```

and, after we introduced addition on naturals,

```
shr(shr(A,M),N) == shr(A,M+N)
```

We began to suspect that `shr` implemented a right shift, and to test this hypothesis we formulated the property

```
prop_shr_value() ->
  ?FORALL({N,A},{nat(),fp()},
          shr(multiply(A,pow(10,N)),N) == A).
```

and after 100,000 successful tests concluded that our hypothesis was correct.

3.4.1 Summing Up

Overall, we found QuickSpec to be a very useful aid in developing an understanding of the fixed point library. Of course, we could simply have formulated the expected equations as QuickCheck properties, and tested them without the aid of QuickSpec. However, this would have taken very much longer, and because the work is fairly tedious, there is a risk that we might have forgotten to include some important properties. QuickSpec automates the tedious part, and allowed us to spot missing equations quickly.

Of course, QuickSpec also generates *unexpected* equations, and these would be much harder to find using QuickCheck. In particular, when investigating functions such as `adjust`, where we initially had little idea of what they were intended to do, then it would have been very difficult to formulate candidate QuickCheck properties in advance.

Although QuickSpec can run given any signature, we discovered that if QuickSpec is used without sufficient thought, the result is often disappointing. We needed to use our ingenuity to extend the signature with useful auxiliaries, such as `negate`, `precision`, `+` and `max`, to get the best out of QuickSpec.

Since QuickSpec is unsound, it may generate equations which are not true, as it did once in our case study. However, even these false equations can be quite informative, since they are *nearly* true—they are simple statements which passed a few hundred test cases. They are thus likely *misconceptions* about the code, and formulating them, then discovering their falsehood by more extensive testing, contributes in itself to understanding of the code. ("You might think that such-and-such holds, but oh no, consider this case!"). We regularly include such *negative properties* in QuickCheck specifications, to prevent the same misconception arising again. QuickSpec runs relatively few tests of each equation (several hundred), and so, once the most interesting equations have been selected, then it is valuable to QuickCheck them many more times to make sure that they are true. It can also be worthwhile, just as in random testing in general, to tweak the test data generator to get a better distribution of random data.

QuickSpec's equation filtering mostly did a fine job of reducing the number of generated equations. However, it sometimes helped to alter the signature to make QuickSpec's job easier—such as replacing `sub` by the simpler function `negate`.

The case study highlights the difficulties that partial functions can cause: our requirement that the left and right-hand sides of an equation must raise exceptions in *exactly* the same cases leads QuickSpec to generate impenetrable equations containing complex terms whose only purpose is to raise an exception in the right cases. QuickSpec cannot currently find equations that hold, provided preconditions are fulfilled. It would be useful to report "weak equations" too, whose left and right-hand sides are equal whenever both are defined. However, it is not clear how QuickSpec should prune such equations, since "weak equality" is not an equivalence relation and the reasoning principles for "weak equations" are not obvious. At the very least, QuickSpec should inform us when it discovers that a function is partial.

Another way to address partiality would be to generate conditional equations with the function preconditions as the condition. Indeed, this would be a generally useful extension, and the case study also highlights other examples where conditional equations would be useful.

4 Related Work

As mentioned earlier, the existing work that is most similar to ours is Henkel et al. [2007]; a tool for discovering algebraic specifications from Java classes. They generate terms and evaluate them, dynamically identify terms which are equal, then generate equations and filter away redundant ones. There are differences in the kind of equations that can be generated, which have been discussed earlier.

The most important difference in the two approaches is the fact that they start

by generating a large set of *ground* terms when searching for equations, which they then test, filter, generalize and prune. So, their initial set both represents the possible terms that can occur in the equations, and the "test cases" that are run. The term set they use thus becomes extremely large, and in order to control its size, they use heuristics such as only generating random subsets of all possible terms, and restricting values to very small domains. This choice not only sacrifices completeness, but also predictability and controllability. In contrast, we always generate *all* possible terms that can occur in equations (keeping completeness), and then use random testing to gather knowledge about these terms. If we end up with too few equations, we can increase the number of terms; if we end up too many equations, we can increase the number of random tests.

There are other differences as well. They test terms for operational equivalence, which is quite expensive; we use fast structural equivalence or a user-specified equality test. They use a heuristic term-rewriting method for pruning equations which will not handle structural properties well (we note that their case studies do not include commutative and associative operators, which we initially found to be extremely problematic); we use a predictable congruence closure algorithm. We are able to generate equations relating higher-order functions; working in Java, this was presumably not possible. They observe—as we do—that conditional equations would be useful, but neither tool generates them. Our tool appears to be faster (our examples take seconds to run, while comparable examples in their setting take hours). It is unfortunately rather difficult to make a fair comparison between the efficacy and performance of the two approaches, because their tool and examples are not available for download.

Daikon [Ernst et al., 2007] is a tool for inferring likely invariants in C, C++, Java or Perl programs. Daikon observes program variables at selected program points during testing, and applies machine learning techniques to discover relationships between them. For example, Daikon can discover linear relationships between integer variables, such as array indices. Agitar's commercial tool based on Daikon generates test cases for the code under analysis automatically [Boshernitsan et al., 2006]. However, Daikon will not discover, for example, that `reverse(reverse(Xs)) == Xs`, unless such a double application of `reverse` appears in the program under analysis. Whereas Daikon discovers invariants that hold at existing program points, QuickSpec discovers equations between arbitrary terms constructed using an API. This is analogous to the difference between *assertions* placed in program code, and the kind of *properties* which QuickCheck tests, that also invoke the API under test in interesting ways. While Daikon's approach is ideal for imperative code, especially code which loops over arrays, QuickSpec is perhaps more appropriate for analysing pure functions.

Inductive logic programming (ILP) [Muggleton and de Raedt, 1994] aims to infer logic programs from examples—specific instances—of their behaviour. The user provides both a collection of true statements and a collection of false statements, and the ILP tool finds a program consistent with those statements. Our approach only uses *false* statements as input (inequality is established by testing), and is optimized for deriving equalities.

In the area of *Automated Theorem Discovery* (ATD), the aim is to emulate the

human theorem discovery process. The idea can be applied to many different fields, such as mathematics, physics, but also formal verification. An example of an ATD system for mathematicians is MathSaid [McCasland and Bundy, 2006]. The system starts by generating a finite set of *hypotheses*, according to some syntactical rules that capture typical mathematical thinking, for example: if we know $A \Rightarrow B$, we should also check if $B \Rightarrow A$, and if not, under what conditions this holds. Theorem proving techniques are used to select theorems and patch non-theorems. Since this leads to many theorems, a filtering phase decides if theorems are interesting or not, according to a number of different predefined "tests". One such test is the simplicity test, which compares theorems for simplicity based on their proofs, and only keeps the simplest theorems. The aim of their filtering is quite different from ours (they want to filter out theorems that mathematicians would have considered trivial), but the motivation is the same; there are too many theorems to consider.

QuickCheck is our own tool for random testing of functional programs, originally for Haskell [Claessen and Hughes, 2000] and now in a commercial version for Erlang [Arts et al., 2006]. QuickCheck tests *properties* such as the equations that QuickSpec discovers, so one application for QuickSpec is to quickly generate a QuickCheck test suite. However, QuickCheck supports a more general property language, including conditional properties and specifications for functions with side-effects [Claessen and Hughes, 2002, Hughes, 2007]. Both implementations of QuickSpec use QuickCheck to generate random test data; this allows users to exert fine control over the selection of test data by specifying an appropriate QuickCheck generator.

5 Conclusions and Future Work

We have presented a new tool, QuickSpec, which can automatically generate algebraic specifications for functional programs. Although simple, it is remarkably powerful. It can be used to aid program understanding, or to generate a QuickCheck test suite to detect changes in specification as the code under test evolves. We are hopeful that it will enable more users to overcome the barrier that formulating properties can present, and discover the benefits of QuickCheck-style specification and testing.

For future work, we plan to generate conditional equations. In some sense, these can be encoded in what we already have by specifying new custom types with appropriate operators. For example, if we want `x<=y` to occur as a precondition, we might introduce a type `AscPair` of "pairs with ascending elements", and add the functions

```
smaller, larger :: AscPair -> Int
```

and the variable `p :: AscPair` to the API. A conditional equation we could then generate is:

```
isSorted (smaller p : larger p : xs) ==
  isSorted (larger p : xs)
```

(Instead of the perhaps more readable

`x<=y ==> isSorted (x:y:xs) == isSorted (y:xs)`.) But we are still investigating the limitations and applicability of this approach.

Another class of equations we are looking at is equations between program fragments that can have side effects. Our idea is to represent a program fragment by a monadic expression, or similar, and use QuickSpec's existing functionality to derive laws for these fragments. We have a prototype implementation of this but more work is needed.

The Erlang version of QuickSpec uses structural equality in the generated equations, which means that terms that may evaluate to different representations of the same abstract value are considered to be different, for example causing some of the unexpected results in section 3.2. The Haskell version uses the `(==)` operator, defined in the appropriate `Eq` instance. However, this is unsafe unless `(==)` is a congruence relation with respect to the operations in the API under test! QuickSpec has recently been extended to *test* for these properties while classifying terms, although we do not discuss this here.

It can be puzzling when an equation we expect to see is *missing*. A small extension would be to allow us to *ask* QuickSpec why an equation wasn't printed, and get either a proof of the equation (if it was pruned away) or a counterexample (if it was false). Both of these can quite easily be extracted from QuickSpec's data structures.

Paper II

Finding Race Conditions in Erlang with QuickCheck and PULSE

This paper was presented at ICFP 2009 in Edinburgh.

Finding Race Conditions in Erlang with QuickCheck and PULSE

Koen Claessen, Michał Pałka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, Ulf Wiger

Abstract

We address the problem of testing and debugging concurrent, distributed Erlang applications. In concurrent programs, race conditions are a common class of bugs and are very hard to find in practice. Traditional unit testing is normally unable to help finding all race conditions, because their occurrence depends so much on timing. Therefore, race conditions are often found during system testing, where due to the vast amount of code under test, it is often hard to diagnose the error resulting from race conditions. We present three tools (QuickCheck, PULSE, and a visualizer) that in combination can be used to test and debug concurrent programs in unit testing with a much better possibility of detecting race conditions. We evaluate our method on an industrial concurrent case study and illustrate how we find and analyze the race conditions.

1 Introduction

Concurrent programming is notoriously difficult, because the non-deterministic interleaving of events in concurrent processes can lead software to work most of the time, but fail in rare and hard-to-reproduce circumstances when an unfortunate order of events occurs. Such failures are called *race conditions*. In particular, concurrent software may work perfectly well during *unit testing*, when individual modules (or "software units") are tested in isolation, but fail later on during *system testing*. Even if unit tests cover all aspects of the units, we still can detect concurrency errors when all components of a software system are tested together. Timing delays caused by other components lead to new, previously untested, schedules of actions performed by the individual units. In the worst case, bugs may not appear until the system is put under heavy load in production. Errors discovered in these late stages are far more expensive to diagnose and correct, than errors found during unit testing. Another cause of concurrency errors showing up at a late stage is when well-tested software is ported from a single-core to a multi-core processor. In that case, one would really benefit from a hierarchical approach to testing legacy code in order to simplify debugging of faults encountered.

The Erlang programming language [Armstrong, 2007] is designed to simplify concurrent programming. Erlang processes do not share memory, and Erlang data structures are immutable, so the kind of *data races* which plague imperative programs, in which concurrent processes race to read and write the same memory location, simply cannot occur. However, this does not mean that Erlang programs are immune to race conditions. For example, the order in which messages are delivered to a process may be non-deterministic, and an unexpected order may lead to failure. Likewise, Erlang processes can share *data*, even if they do not share memory—the file store is one good example of shared

mutable data, but there are also shared data-structures managed by the Erlang virtual machine, which processes can race to read and write.

Industrial experience is that the late discovery of race conditions is a real problem for Erlang developers too [Cronqvist, 2004]. Moreover, these race conditions are often caused by design errors, which are particularly expensive to repair. If these race conditions could be found during unit testing instead, then this would definitely reduce the cost of software development.

In this paper, we describe tools we have developed for finding race conditions in Erlang code during unit testing. Our approach is based on *property-based testing* using QuickCheck [Claessen and Hughes, 2000], in a commercial version for Erlang developed by Quviq AB [Arts et al., 2006, Hughes, 2007]. Its salient features are described in section 3. We develop a suitable property for testing parallel code, and a method for generating parallel test cases, in section 4. To test a wide variety of schedules, we developed a randomizing scheduler for Erlang called PULSE, which we explain in section 5. PULSE records a trace during each test, but interpreting the traces is difficult, so we developed a trace visualizer which is described in section 6. We evaluate our tools by applying them to an industrial case study, which is introduced in section 2, then used as a running example throughout the paper. This code was already known to contain bugs (thanks to earlier experiments with QuickCheck in 2005), but we were previously unable to *diagnose* the problems. Using the tools described here, we were able to find and fix two race conditions, and identify a fundamental flaw in the API.

2 Our case study: the process registry

We begin by introducing the industrial case that we apply our tools and techniques to. In Erlang, each process has a unique, dynamically-assigned identifier ("pid"), and to send a message to a process, one must know its pid. To enable processes to discover the pids of central services, such as error logging, Erlang provides a *process registry*—a kind of local name server—which associates static names with pids. The Erlang VM provides operations to *register* a pid with a name, to *look up* the pid associated with a name, and to *unregister* a name, removing any association with that name from the registry. The registry holds only *live* processes; when registered processes crash, then they are automatically unregistered. The registry is heavily used to provide access to system services: a newly started Erlang node already contains 13 registered processes.

However, the built-in process registry imposes several, sometimes unwelcome, limitations: registered names are restricted to be atoms, the same process cannot be registered with multiple names, and there is no efficient way to search the registry (other than by name lookup). This motivated Ulf Wiger (who was working for Ericsson at the time) to develop an extended process registry *in Erlang*, which could be modified and extended much more easily than the one in the virtual machine. Wiger's process registry software has been in use in Ericsson products for several years [Wiger, 2007].

In our case study we consider an earlier prototype of this software, called `proc_reg`, incorporating an optimization that proved not to work. The API

supported is just: `reg(Name,Pid)` to register a pid, `where(Name)` to look up a pid, `unreg(Name)` to remove a registration, and finally `send(Name,Msg)` to send a message to a registered process. Like the production code, `proc_reg` stores the association between names and pids in *Erlang Term Storage* ("ETS tables")—hash tables, managed by the virtual machine, that hold a set of tuples and support tuple-lookup using the first component as a key [cf. Armstrong, 2007, chap 15]. It also creates a *monitor* for each registered process, whose effect is to send `proc_reg` a "`DOWN`" message if the registered process crashes, so it can be removed from the registry. Two ETS table entries are created for each registration: a "forward" entry that maps names to pids, and a "reverse" entry that maps registered pids to the monitor reference. The monitor reference is needed to turn off monitoring again, if the process should later be unregistered.

Also like the production code, `proc_reg` is implemented as a server process using Erlang's *generic server* library [cf. Armstrong, 2007, chap 16]. This library provides a robust way to build client-server systems, in which clients make "synchronous calls" to the server by sending a `call` message, and awaiting a matching reply[1]. Each operation—`reg`, `where`, `unreg` and `send`—is supported by a different `call` message. The operations are actually executed by the server, one at a time, and so no race conditions can arise.

At least, this is the theory. In practice there is a small cost to the generic server approach: each request sends two messages and requires two context switches, and although these are cheap in Erlang, they are not free, and turn out to be a bottleneck in system start-up times, for example. The prototype `proc_reg` attempts to optimize this, by moving the creation of the first "forward" ETS table entry into the clients. If this succeeds (because there is no previous entry with that name), then clients just make an "asynchronous" call to the server (a so-called `cast` message, with no reply) to inform it that it should complete the registration later. This avoids a context switch, and reduces two messages to one. If there *is* already a registered process with the same name, then the `reg` operation fails (with an exception)—unless, of course, the process is dead. In this case, the process will soon be removed from the registry by the server; clients ask the server to "audit" the dead process to hurry this along, then complete their registration as before.

This prototype was one of the first pieces of software to be tested using QuickCheck at Ericsson. At the time, in late 2005, it was believed to work, and indeed was accompanied by quite an extensive suite of unit tests—including cases designed specifically to test for race conditions. We used QuickCheck to generate and run random sequences of API calls in two concurrent processes, and instrumented the `proc_reg` code with calls to `yield()` (which cedes control to the scheduler) to cause fine-grain interleaving of concurrent operations. By so doing, we could show that `proc_reg` was incorrect, since our tests failed. But the failing test cases we found were large, complex, and very hard to understand, and we were unable to use them to diagnose the problem. As a result, this version of `proc_reg` was abandoned, and development of the production version continued without the optimization.

While we were pleased that QuickCheck could reveal bugs in `proc_reg`, we were

[1] Unique identifiers are generated for each call, and returned in the reply, so that no message confusion can occur.

unsatisfied that it could not help us to find them. Moreover, the QuickCheck property we used to test it was hard-to-define and ad hoc—and not easily reusable to test any other software. This paper is the story of how we addressed these problems—and returned to apply our new methods successfully to the example that defeated us before.

3 An Overview of Quviq QuickCheck

QuickCheck [Claessen and Hughes, 2000] is a tool that tests universally quantified *properties*, instead of single test cases. QuickCheck generates random test cases from each property, tests whether the property is true in that case, and reports cases for which the property fails. Recent versions also "shrink" failing test cases automatically, by searching for similar, but smaller test cases that also fail. The result of shrinking is a "minimal"[2] failing case, which often makes the root cause of the problem very easy to find.

Quviq QuickCheck is a commercial version that includes support for model-based testing using a state machine model [Hughes, 2007]. This means that it has standard support for generating sequences of API calls using this state machine model. It has been used to test a wide variety of industrial software, such as Ericsson's Media Proxy [Arts et al., 2006] among others. State machine models are tested using an additional library, `eqc_statem`, which invokes call-backs supplied by the user to generate and test random, well-formed sequences of calls to the software under test. We illustrate `eqc_statem` by giving fragments of a (sequential) specification of `proc_reg`.

Let us begin with an example of a generated test case (a sequence of API calls).

```
[{set,{var,1},{call,proc_reg_eqc,spawn,[]}},
 {set,{var,2},{call,proc_reg,where,[c]}},
 {set,{var,3},{call,proc_reg_eqc,spawn,[]}},
 {set,{var,4},{call,proc_reg_eqc,kill,[{var,1}]}},
 {set,{var,5},{call,proc_reg,where,[d]}},
 {set,{var,6},{call,proc_reg_eqc,reg,[a,{var,1}]}},
 {set,{var,7},{call,proc_reg_eqc,spawn,[]}}]
```

`eqc_statem` test cases are lists of *symbolic commands* represented by Erlang terms, each of which binds a symbolic variable (such as `{var,1}`) to the result of a function call, where `{call,M,F,Args}` represents a call of function `F` in module `M` with arguments `Args`[3]. Note that previously bound variables can be used in later calls. Test cases for `proc_reg` in particular randomly spawn processes (to use as test data), kill them (to simulate crashes at random times), or pass them to `proc_reg` operations. Here `proc_reg_eqc` is the module containing the specification of `proc_reg`, in which we define local versions of `reg` and `unreg` which just call `proc_reg` and catch any exceptions. This allows us to write properties that test whether an exception is raised correctly or not. (An *uncaught* exception in a test is interpreted as a failure of the entire test).

We model the state of a test case as a list of processes spawned, processes killed, and the `{Name,Pid}` pairs currently in the registry. We normally encapsulate

[2] In the sense that it cannot shrink to a failing test with the shrinking algorithm used.

[3] In Erlang, variables start with an uppercase character, whereas atoms (constants) start with a lowercase character.

the state in a record:

```
-record(state,{pids=[],regs=[],killed=[]}).
```

`eqc_statem` generates random calls using the call-back function `command` that we supply as part of the state machine model, with the test case state as its argument:

```
command(S) ->
  oneof(
    [{call,?MODULE,spawn,[]}] ++
    [{call,?MODULE,kill, [elements(S#state.pids)]}
           || S#state.pids/=[]] ++
    [{call,?MODULE,reg,[name(),elements(S#state.pids)]}
           || S#state.pids/=[]] ++
    [{call,?MODULE,unreg,[name()]}] ++
    [{call,proc_reg,where,[name()]}]).

name() -> elements([a,b,c,d]).
```

The function `oneof` is a QuickCheck generator that randomly uses one element from a list of generators; in this case, the list of candidates to choose from depends on the test case state. (`[X||P]` is a degenerate list comprehension, that evaluates to the empty list if `P` is false, and `[X]` if `P` is true—so `reg` and `kill` can be generated only if there are pids available to pass to them.) We decided not to include `send` in test cases, because its implementation is quite trivial. The macro `?MODULE` expands to the name of the module that it appears in, `proc_reg_eqc` in this case.

The `next_state` function specifies how each call is supposed to change the state:

```
next_state(S,V,{call,_,spawn,_}) ->
  S#state{pids=[V|S#state.pids]};
next_state(S,V,{call,_,kill,[Pid]}) ->
  S#state{killed=[Pid|S#state.killed],
          regs=[{Name,P} ||
                {Name,P} <- S#state.regs, Pid /= P]};
next_state(S,_V,{call,_,reg,[Name,Pid]}) ->
  case register_ok(S,Name,Pid) andalso
         not lists:member(Pid,S#state.killed) of
       true ->
         S#state{regs=[{Name,Pid}|S#state.regs]};
       false ->
         S
  end;
next_state(S,_V,{call,_,unreg,[Name]}) ->
  S#state{regs=lists:keydelete(Name,1,S#state.regs)};
next_state(S,_V,{call,_,where,[_]}) ->
  S.

register_ok(S,Name,Pid) ->
  not lists:keymember(Name,1,S#state.regs).
```

Note that the new state can depend on the *result* of the call (the second argument `V`), as in the first clause above. Note also that killing a process removes it from the registry (in the model), and that registering a dead process, or a name that is already registered (see `register_ok`), should not change the

registry state. We do allow the same pid to be registered with several names, however.

When running tests, `eqc_statem` checks the postcondition of each call, specified via another call-back that is given the state before the call, and the actual result returned, as arguments. Since we catch exceptions in each call, which converts them into values of the form `{'EXIT',Reason}`, our `proc_reg` postconditions can test that exceptions are raised under precisely the right circumstances:

```
postcondition(S,{call,_,reg,[Name,Pid]},Res) ->
  case Res of
       true ->
         register_ok(S,Name,Pid);
       {'EXIT',_} ->
         not register_ok(S,Name,Pid)
  end;
postcondition(S,{call,_,unreg,[Name]},Res) ->
  case Res of
       true ->
         unregister_ok(S,Name);
       {'EXIT',_} ->
         not unregister_ok(S,Name)
  end;
postcondition(S,{call,_,where,[Name]},Res) ->
  lists:member({Name,Res},S#state.regs);
postcondition(_S,{call,_,_,_},_Res) ->
  true.

unregister_ok(S,Name) ->
  lists:keymember(Name,1,S#state.regs).
```

Note that `reg(Name,Pid)` and `unreg(Name)` are required to return exceptions if `Name` is already used/not used respectively, but that `reg` always returns `true` if `Pid` is dead, even though no registration is performed! This may perhaps seem a surprising design decision, but it is consistent. As a comparison, the built-in process registry sometimes returns `true` and sometimes raises an exception when registering dead processes. This is due to the fact that a context switch is required to clean up.

State machine models can also specify a *precondition* for each call, which restricts test cases to those in which all preconditions hold. In this example, we could have used preconditions to exclude test cases that we expect to raise exceptions—but we prefer to allow any test case, and check that exceptions are raised correctly, so we define all preconditions to be `true`.

With these four call-backs, plus another call-back specifying the initial state, our specification is almost complete. It only remains to define the top-level property which generates and runs tests:

```
prop_proc_reg() ->
  ?FORALL(Cmds,commands(?MODULE),
     begin
       {ok,ETSTabs} = proc_reg_tabs:start_link(),
       {ok,Server} = proc_reg:start_link(),
       {H,S,Res} = run_commands(?MODULE,Cmds),
       cleanup(ETSTabs,Server),
       Res == ok
     end).
```

Here `?FORALL` binds `Cmds` to a random list of commands generated by `commands`, then we initialize the registry, run the commands, clean up, and check that the result of the run (`Res`) was a success. Here `commands` and `run_commands` are provided by `eqc_statem`, and take the current module name as an argument in order to find the right call-backs. The other components of `run_commands`' result, `H` and `S`, record information about the test run, and are of interest primarily when a test fails. This is not the case here: *sequential* testing of `proc_reg` does not fail.

4 Parallel Testing with QuickCheck

4.1 A Parallel Correctness Criterion

In order to test for race conditions, we need to generate test cases that are executed in parallel, and we also need *a specification of the correct parallel behavior*. We have chosen, in this paper, to use a specification that just says that *the API operations we are testing should behave atomically.*

How can we tell from test results whether or not each operation "behaved atomically"? Following Lamport [1979] and Herlihy and Wing [1987], we consider a test to have passed if the observed results are the same as some possible sequential execution of the operations in the test—that is, a possible interleaving of the parallel processes in the test.

Of course, testing for atomic behavior is just a special case, and in general we may need to test other properties of concurrent code too—but we believe that this is a very important special case. Indeed, Herlihy and Wing claim that their notion of *linearizability* "focuses exclusively on a subset of concurrent computations that we believe to be the most interesting and useful"; we agree. In particular, atomicity is of great interest for the process registry.

One great advantage of this approach is that we can reuse the *same* specification of the sequential behavior of an API, to test its behavior when invocations take place in parallel. We need only find the right linearization of the API calls in the test, and then use the sequential specification to determine whether or not the test has passed. We have implemented this idea in a new QuickCheck module, `eqc_par_statem`, which takes the *same* state-machine specifications as `eqc_statem`, but tests the API in parallel instead. While state machine specifications require some investment to produce in real situations, this means that we can test for race conditions *with no further investment* in developing a parallel specification. It also means that, as the code under test evolves, we can switch freely to-and-fro between sequential testing to ensure the basic behavior still works, and race condition testing using `eqc_par_statem`.

The difficulty with this approach is that, when we run a test, then there is no way to *observe* the sequence in which the API operations take effect. (For example, a server is under no obligation to service requests in the order in which they are made, so observing this order would tell us nothing.) In general, the only way to tell whether there is a possible sequentialization of a test case which can explain the observed test results, is to *enumerate* all possible sequentializations. This is prohibitively expensive unless care is taken when

test cases are generated.

4.2 Generating Parallel Test Cases

Our first approach to parallel test case generation was to use the standard Quviq QuickCheck library `eqc_statem` to generate sequential test cases, then execute all the calls in the test case in parallel, constrained only by the data dependencies between them (which arise from symbolic variables, bound in one command, being used in a later one). This generates a great deal of parallelism, but sadly also an enormous number of possible serializations—in the worst case in which there are no data dependencies, a sequence of n commands generates $n!$ possible serializations. It is not practically feasible to implement a test oracle for parallel tests of this sort.

Instead, we decided to generate parallel test cases of a more restricted form. They consist of an initial sequential prefix, to put the system under test into a random state, followed by exactly *two* sequences of calls which are performed in parallel. Thus the possible serializations consist of the initial prefix, followed by an interleaving of the two parallel sequences. (Lu et al. [2008] gives clear evidence that it is possible to discover a large fraction of the concurrency related bugs by using only two parallel threads/processes.) We generate parallel test cases by parallelizing a suffix of an `eqc_statem` test case, separating it into two lists of commands of roughly equal length, with no mutual data dependencies, which are *non-interfering* according to the sequential specification. By non-interference, we mean that all command preconditions are satisfied in any interleaving of the two lists, which is necessary to prevent tests from failing because a precondition was unsatisfied—not an interesting failure. We avoid parallelizing too long a suffix (longer than 16 commands), to keep the number of possible interleavings feasible to enumerate (about 10,000 in the worst case). Finally, we run tests by first running the prefix, then spawning two processes to run the two command-lists in parallel, and collecting their results, which will be non-deterministic depending on the actual parallel scheduling of events.

We decide whether a test has passed, by attempting to construct a sequentialization of the test case which explains the results observed. We begin with the sequential prefix of the test case, and use the `next_state` function of the `eqc_statem` model to compute the test case state after this prefix is completed. Then we try to *extend* the sequential prefix, one command at a time, by choosing the first command from one of the parallel branches, and moving it into the prefix. This is allowed only if the `postcondition` specified in the `eqc_statem` model accepts the actual result returned by the command when we ran the test. If so, we use the `next_state` function to compute the state after this command, and continue. If the first commands of *both* branches fulfilled their postconditions, then we cannot yet determine which command took effect first, and we must explore both possibilities further. If we succeed in moving *all* commands from the parallel branches into the sequential prefix, such that all postconditions are satisfied, then we have found a possible sequentialization of the test case explaining the results we observed. If our search fails, then there is *no* such sequence, and the test failed.

This is a greedy algorithm: as soon as a postcondition fails, then we can discard

all potential sequentializations with the failing command as the next one in the sequence. This happens often enough to make the search reasonably fast in practice. As a further optimization, we memoize the search function on the remaining parallel branches and the test case state. This is useful, for example, when searching for a sequentialization of $[A, B]$ and $[C, D]$, if both $[A, C]$ and $[C, A]$ are possible prefixes, and they lead to the same test state—for then we need only try to sequentialize $[B]$ and $[D]$ once. We memoize the non-interference test in a similar way, and these optimizations give an appreciable, though not dramatic, speed-up in our experiments—of about 20%. With these optimizations, generating and running parallel tests is acceptably fast.

4.3 Shrinking Parallel Test Cases

When a test fails, QuickCheck attempts to *shrink* the failing test by searching for a similar, but smaller test case which also fails. QuickCheck can often report minimal failing examples, which is a great help in fault diagnosis. `eqc_statem` already has built-in shrinking methods, of which the most important tries to delete unnecessary commands from the test case, and `eqc_par_statem` inherits these methods. But we also implement an additional shrinking method for parallel test cases: if it is possible to move a command from one of the parallel suffixes into the sequential prefix, then we do so. Thus the minimal test cases we find are "minimally parallel"—we know that the parallel branches in the failing tests reported really do race, because everything that can be made sequential, is sequential. This also assists fault diagnosis.

4.4 Testing `proc_reg` for Race Conditions

To test the process registry using `eqc_par_statem`, it is only necessary to modify the property in Section 2 to use `eqc_par_statem` rather than `eqc_statem` to generate and run test cases.

```
prop_proc_reg_parallel() ->
  ?FORALL(Cmds,eqc_par_statem:commands(?MODULE),
     begin
       {ok,ETSTabs} = proc_reg_tabs:start_link(),
       {ok,Server} = proc_reg:start_link(),
       {H,{A,B},Res} =
         eqc_par_statem:run_commands(?MODULE,Cmds),
       cleanup(ETSTabs,Server),
       Res == ok
     end).
```

The type returned by `run_commands` is slightly different (`A` and `B` are lists of the calls made in each parallel branch, paired with the results returned), but otherwise no change to the property is needed.

When this property is tested on a single-core processor, all tests pass. However, as soon as it is tested on a dual-core, tests begin to fail. Interestingly, just running on two cores gives us enough fine-grain interleaving of concurrent processes to demonstrate the presence of race conditions, something we had to achieve by instrumenting the code with calls to `yield()` to control the scheduler when we first tested this code in 2005. However, just as in 2005, the

reported failing test cases are large, and do not shrink to small examples. This makes the race condition very hard indeed to diagnose.

The problem is that the test outcome is not determined solely by the test case: depending on the actual interleaving of memory operations on the dual core, the same test may sometimes pass and sometimes fail. This is devastating for QuickCheck's shrinking, which works by repeatedly replacing the failed test case by a smaller one which still fails. If the smaller test happens to succeed—by sheer chance, as a result of non-deterministic execution—then the shrinking process stops. This leads QuickCheck to report failed tests which are far from minimal.

Our solution to this is almost embarrassing in its simplicity: instead of running each test only once, we run it many times, and consider a test case to have passed only if it passes every time we run it. We express this concisely using a new form of QuickCheck property, `?ALWAYS(N,Prop)`, which passes if `Prop` passes `N` times in a row[4]. Now, provided the race condition we are looking for is reasonably likely to be provoked by test cases in which it is present, then `?ALWAYS(10,...)` is very likely to provoke it—and so tests are unlikely to succeed "by chance" during the shrinking process. This dramatically improves the effectiveness of shrinking, even for quite small values of `N`. While we do not *always* obtain minimal failing tests with this approach, we find we can usually obtain a minimal example by running QuickCheck a few times.

When testing the `proc_reg` property above, we find the following simple counterexample:

```
{[{set,{var,5},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,9},{call,proc_reg_eqc,kill,[{var,5}]}},
  {set,{var,15},{call,proc_reg_eqc,reg,[a,{var,5}]}}],
 {[{set,{var,19},{call,proc_reg_eqc,reg,[a,{var,5}]}}],
  [{set,{var,18},{call,proc_reg_eqc,reg,[a,{var,5}]}}]}}
```

This test case first creates and kills a process, then tries to register it (which should have no effect, because it is already dead), and finally tries to register it again twice, in parallel. Printing the diagnostic output from `run_commands`, we see:

```
Sequential:
  [{{state,[],[],[]},<0.5576.2>},
   {{state,[<0.5576.2>],[],[]},ok},
   {{state,[<0.5576.2>],[],[<0.5576.2>]},true}]
Parallel:
  {[{{call,proc_reg_eqc,reg,[a,<0.5576.2>]},
     {'EXIT',{badarg,[{proc_reg,reg,2},...]}}}],
   [{{call,proc_reg_eqc,reg,[a,<0.5576.2>]},true}]}
Res: no_possible_interleaving
```

(where the ellipses replace an uninteresting stack trace). The values displayed under "`Parallel:`" are the results `A` and `B` from the two parallel branches—they reveal that one of the parallel calls to `reg` raised an exception, even though trying to register a dead process should always just return `true`! How this happened, though, is still quite mysterious—but will be explained in the following sections.

[4] In fact we need only repeat tests during shrinking.

5 PULSE: A User-level Scheduler

At this point, we have found a simple test case that fails, but we do not know *why* it failed—we need to debug it. A natural next step would be to turn on Erlang's tracing features and rerun the test. But when the bug is caused by a race condition, then turning on tracing is likely to change the timing properties of the code, and thus interfere with the test failure! Even simply repeating the test may lead to a different result, because of the non-determinism inherent in running on a multi-core. This is devastating for debugging.

What we need is to be able to *repeat* the test as many times as we like, with deterministic results, and to *observe* what happens during the test, so that we can analyze how the race condition was provoked. With this in mind, we have implemented a new Erlang module that can control the execution of designated Erlang processes and records a trace of all relevant events. Our module can be thought of as a *user-level scheduler*, sitting on top of the normal Erlang scheduler. Its aim is to take control over all sources of non-determinism in Erlang programs, and instead take those scheduling decisions randomly. This means that we can repeat a test using exactly the same schedule by supplying the same random number seed: this makes tests repeatable. We have named the module PULSE, short for *ProTest* User-Level Scheduler for Erlang.

The Erlang virtual machine (VM) runs processes for relatively long time-slices, in order to minimize the time spent on context switching—but as a result, it is very unlikely to provoke race conditions in small tests. It is possible to tune the VM to perform more context switches, but even then the scheduling decisions are entierly deterministic. This is one reason why tricky concurrency bugs are rarely found during unit testing; it is not until later stages of a project, when many components are tested together, that the standard scheduler begins to preempt processes and trigger race conditions. In the worst case, bugs don't appear until the system is put under heavy load in production! In these later stages, such errors are expensive to debug. One other advantage (apart from repeatability) of PULSE is that it generates much more fine-grain interleaving than the built-in scheduler in the Erlang virtual machine (VM), because it randomly chooses the next process to run at each point. Therefore, we can provoke race conditions even in very small tests.

Erlang's scheduler is built into its virtual machine—and we did *not* want to modify the virtual machine itself. Not only would this be difficult—it is a low-level, fairly large and complex C program—but we would need to repeat the modifications every time a new version of the virtual machine was released. We decided, therefore, to implement PULSE in Erlang, as a user-level scheduler, and to *instrument* the code of the processes that it controls so that they cooperate with it. As a consequence, PULSE can even be used in conjunction with legacy or customized versions of the Erlang VM (which are used in some products). The user level scheduler also allows us to restrict our debugging effort to a few processes, whereas we are guaranteed that the rest of the processes are executed normally.

5.1 Overall Design

The central idea behind developing PULSE was to provide absolute control over the order of relevant events. The first natural question that arises is: What are the relevant events? We define a *side-effect* to be any interaction of a process with its environment. Of particular interest in Erlang is the way processes interact by message passing, which is asynchronous. Message channels, containing messages that have been sent but not yet delivered, are thus part of the environment and explicitly modelled as such in PULSE. It makes sense to separate side-effects into two kinds: *outward* side-effects, that influence only the environment (such as sending a message over a channel, which does not block and cannot fail, or printing a message), and *inward* side-effects, that allow the environment to influence the behavior of the process (such as receiving a message from a channel, or asking for the system time).

We do not want to take control over purely functional code, or side-effecting code that only influences processes locally. PULSE takes control over some basic features of the Erlang RTS (such as spawning processes, message sending, linking, etc.), but it knows very little about standard library functions – it would be too much work to deal with each of these separately! Therefore, the user of PULSE can specify which library functions should be dealt with as (inward) side-effecting functions, and PULSE has a generic way of dealing with these (see subsection 5.3).

A process is only under the control of PULSE if its code has been properly instrumented. All other processes run as normal. In instrumentation, occurrences of side-effecting actions are replaced by indirections that communicate with PULSE instead. In particular, outward side-effects (such as sending a message to another process) are replaced by simply sending a message to PULSE with the details of the side-effect, and inward side-effects (such as receiving a message) are replaced by sending a request to PULSE for performing that side-effect, and subsequently waiting for permission. To ease the instrumentation process, we provide an automatic instrumenter, described in subsection 5.4.

5.2 Inner Workings

The PULSE scheduler controls its processes by allowing only one of them to run at a time. It employs a cooperative scheduling method: At each decision point, PULSE randomly picks one of its waiting processes to proceed, and wakes it up. The process may now perform a number of outward side-effects, which are all recorded and taken care of by PULSE, until the process wants to perform an inward side-effect. At this point, the process is put back into the set of waiting processes, and a new decision point is reached.

The (multi-node) Erlang semantics [Svensson and Fredlund, 2007] provides only one guarantee for message delivery order: that messages between a **pair** of processes arrive in the same order as they were sent. So as to adhere to this, PULSE's state also maintains a message queue between each pair of processes. When process P performs an outward side-effect by sending a message M to the process Q, then M is added to the queue $\langle P, Q\rangle$. When PULSE wants to wake up a waiting process Q, it does so by randomly picking a non-empty

queue $\langle P', Q \rangle$ with Q as its destination, and delivering the first message in that queue to Q. Special care needs to be taken for the Erlang construct `receive ... after` n `-> ... end`, which allows a receiving process to only wait for an incoming message for n milliseconds before continuing, but the details of this are beyond the scope of this paper.

As an additional benefit, this design allows PULSE to detect deadlocks when it sees that all processes are blocked, and there exist no message queues with the blocked processes as destination.

As a clarification, the message queues maintained by PULSE for each pair of processes should not be confused with the internal mailbox that each process in Erlang has. In our model, sending a message M from P to Q goes in four steps: (1) P asynchronously sends off M, (2) M is on its way to Q, (3) M is delivered to Q's mailbox, (4) Q performs a `receive` statement and M is selected and removed from the mailbox. The only two events in this process that we consider side-effects are (1) P sending of M, and (3) delivering M to Q's mailbox. In what order a process decides to process the messages in its mailbox is not considered a side-effect, because no interaction with the environment takes place.

5.3 External Side-effects

In addition to sending and receiving messages between themselves, the processes under test can also interact with uninstrumented code. PULSE needs to be able to control the order in which those interactions take place. Since we are not interested in controlling the order in which pure functions are called we allow the programmer to specify which external functions have side-effects. Each call of a side-effecting function is then instrumented with code that `yield`s before performing the real call and PULSE is free to run another process at that point.

Side-effecting functions are treated as atomic which is also an important feature that aids in testing systems built of multiple components. Once we establish that a component contains no race conditions we can remove the instrumentation from it and mark its operations as atomic side-effects. We will then be able to test other components that use it and each operation marked as side-effecting will show up as a single event in a trace. Therefore, it is possible to test a component for race conditions independently of the components that it relies on.

5.4 Instrumentation

The program under test has to cooperate with PULSE, and the relevant processes should use PULSE's API to send and receive messages, spawn processes, etc., instead of Erlang's built-in functionality. Manually altering an Erlang program so that it does this is tedious and error-prone, so we developed an instrumenting compiler that does this automatically. The instrumenter is used in exactly the same way as the normal compiler, which makes it easy to switch between PULSE and the normal Erlang scheduler. It's possible to instrument and load a module

at runtime by typing in a single command at the Erlang shell.

Let us show the instrumentation of the four most important constructs: sending a message, yielding, spawning a process, and receiving a message.

5.4.1 Sending

If a process wants to send a message, the instrumenter will redirect this as a request to the PULSE scheduler. Thus, `Pid ! Msg` is replaced by

```
scheduler ! {send, Pid, Msg},
Msg
```

The result value of sending a message is always the message that was sent. Since we want the instrumented *send* to yield the same result value as the original one, we add the second line.

5.4.2 Yielding

A process yields when it wants to give up control to the scheduler. Yields are also introduced just before each user-specified side-effecting external function.

After instrumentation, a yielding process will instead give up control to PULSE. This is done by telling it that the process yields, and waiting for permission to continue. Thus, `yield()` is replaced by

```
scheduler ! yield,
receive
  {scheduler, go} -> ok
end
```

In other words, the process notifies PULSE and then waits for the message `go` from the scheduler before it continues. All control messages sent by PULSE to the controlled processes are tagged with `{scheduler, _}` in order to avoid mixing them up with "real" messages.

5.4.3 Spawning

A process P spawning a process Q is considered an outward side-effect for P, and thus P does not have to block. However, PULSE must be informed of the existence of the new process Q, and Q needs to be brought under its control. The spawned process Q must therefore wait for PULSE to allow it to run. Thus, `spawn(Fun)` is replaced by

```
Pid = spawn(fun() -> receive
                        {scheduler, go} -> Fun()
                     end
            end),
scheduler ! {spawned, Pid},
Pid
```

In other words, the process spawns an altered process that waits for the message `go` from the scheduler before it does anything. The scheduler is then informed of the existence of the spawned process, and we continue.

5.4.4 Receiving

Receiving in Erlang works by pattern matching on the messages in the process' mailbox. When a process is ready to receive a new message, it will have to ask PULSE for permission. However, it is possible that an appropriate message already exists in its mailbox, and receiving this message would not be a side-effect. Therefore, an instrumented process will first check if it is possible to receive a message with the desired pattern, and proceed if this is possible. If not, it will tell the scheduler that it expects a new message in its mailbox, and blocks. When woken up again on the delivery of a new message, this whole process is repeated if necessary.
We need a helper function that implements this checking-waiting loop. It is called `receiving`:

```
receiving(Receiver) ->
  Receiver(fun() ->
    scheduler ! block,
    receive
      {scheduler, go} -> receiving(Receiver)
    end
  end).
```

`receiving` gets a receiver function as an argument. A receiver function is a function that checks if a certain message is in its mailbox, and if not, executes its argument function. The function `receiving` turns this into a loop that only terminates once PULSE has delivered the right message. When the receiver function fails, PULSE is notified by the `block` message, and the process waits for permission to try again.
Code of the form

```
receive Pat -> Exp end
```

is then replaced by

```
receiving(fun (Failed) ->
            receive
              Pat     -> Exp
              after 0 -> Failed()
            end
          end)
```

In the above, we use the standard Erlang idiom (`receive` ... `after 0 ->` ... `end`) for checking if a message of a certain type exists. It is easy to see how receive statements with more than one pattern can be adapted to work with the above scheme.

5.5 Testing `proc_reg` with PULSE

To test the `proc_reg` module using both QuickCheck and PULSE, we need to make a few modifications to the QuickCheck property in Section 4.4.

```
prop_proc_reg_scheduled() ->
  ?FORALL(Cmds,eqc_par_statem:commands(?MODULE),
   ?ALWAYS(10,?FORALL(Seed,seed(),
```

```
    begin

      SRes =
        scheduler:start([{seed,Seed}],
          fun() ->
            {ok,ETSTabs} = proc_reg_tabs:start_link(),
            {ok,Server} = proc_reg:start_link(),
            eqc_par_statem:run_commands(?MODULE,Cmds),
            cleanup(ETSTabs,Server),
          end),
      {H,AB,Res} = scheduler:get_result(SRes),
      Res == ok
    end))).
```

PULSE uses a random seed, generated by `seed()`. It also takes a function as an argument, so we create a lambda-function which initializes and runs the tests. The result of running the scheduler is a list of things, thus we need to call `scheduler:get_result` to retrieve the actual result from `run_commands`. We should also remember to *instrument* rather than compile all the involved modules. Note that we still use `?ALWAYS` in order to run the same test data with different random seeds, which helps the shrinking process in finding smaller failing test cases that would otherwise be less likely to fail.

When testing this modified property, we find the following counterexample, which is in fact simpler than the one we found in Section 4.4:

```
{[{set,{var,9},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,10},{call,proc_reg_eqc,kill,[{var,9}]}}],
 {[{set,{var,15},{call,proc_reg_eqc,reg,[c,{var,9}]}}],
  [{set,{var,12},{call,proc_reg_eqc,reg,[c,{var,9}]}}]}}
```

When prompted, PULSE provides quite a lot of information about the test case run and the scheduling decisions taken. Below we show an example of such information. However, it is still not easy to explain the counterexample; in the next section we present a method that makes it easier to understand the scheduler output.

```
 -> <'start_link.Pid1'> calls
    scheduler:process_flag [priority,high]
    returning normal.
 -> <'start_link.Pid1'> sends
    '{call,{attach,<0.31626.0>},
           <0.31626.0>,#Ref<0.0.0.13087>}'
    to <'start_link.Pid'>.
 -> <'start_link.Pid1'> blocks.
*** unblocking <'start_link.Pid'>
    by delivering '{call,{attach,<0.31626.0>},
                         <0.31626.0>,
                         #Ref<0.0.0.13087>}'
    sent by <'start_link.Pid1'>.
...
```

6 Visualizing Traces

PULSE records a complete trace of the interesting events during test execution, but these traces are long, and tedious to understand. To help us interpret them, we have, utilizing the popular GraphViz package [Gansner and North, 1999],

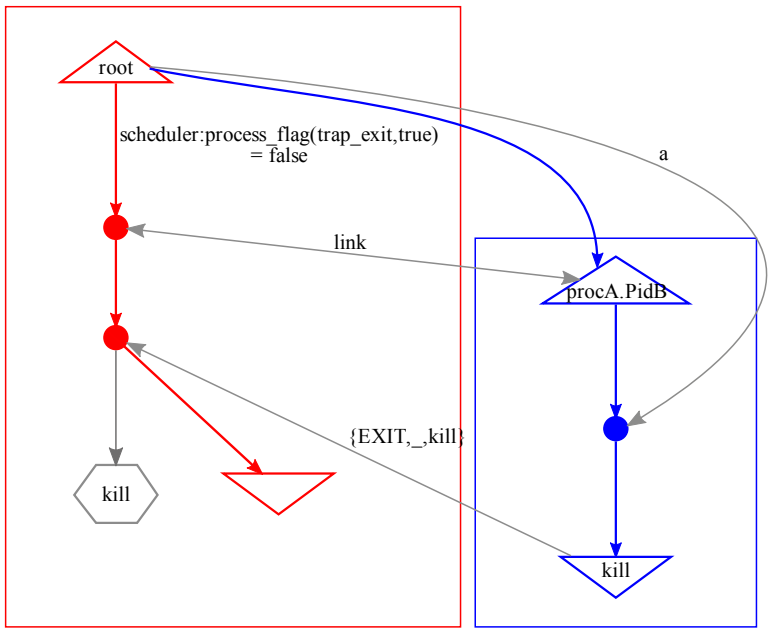

Figure 2.1: A simple trace visualization.

built a *trace visualizer* that draws the trace as a graph. For example, Figure 2.1 shows the graph drawn for one possible trace of the following program:

```
procA() ->
  PidB = spawn(fun procB/0),
  PidB ! a,
  process_flag(trap_exit, true),
  link(PidB),
  receive
    {'EXIT',_,Why} -> Why
  end.

procB() ->
  receive
    a -> exit(kill)
  end.
```

The function `procA` starts by spawning a process, and subsequently sends it a message `a`. Later, `procA` *links* to the process it spawned, which means that it will get notified when that process dies. The default behavior of a process when such a notification happens is to also die (in this way, one can build hierarchies of processes). Setting the process flag *trap_exit* to true changes this behaviour, and the notification is delivered as a regular message of the form `{EXIT,_,_}` instead.

In the figure, each process is drawn as a sequence of state transitions, from a start state drawn as a triangle, to a final state drawn as an inverted triangle, all enclosed in a box and assigned a unique color. (Since the printed version of the diagrams may lack these colors, we reference diagram details by location and not by color. However, the diagrams are even more clear in color.) The diagram shows the two processes, `procA` (called `root`) which is shown to the left (in red), and `procB` (called `procA.PidB`, a name automatically derived by PULSE from the point at which it was spawned) shown to the right (in blue). Message delivery is shown by gray arrows, as is the return of a result by the `root` process. As explained in the previous section, processes make transitions when receiving a message[5], or when calling a function that the instrumenter knows has a side-effect. From the figure, we can see that the `root` process spawned `PidB` and sent the message `a` to it, but before the message was delivered then the `root` process managed to set its `trap_exit` process flag, and linked to `PidB`. `PidB` then received its message, and killed itself, terminating with reason `kill`. A message was sent back to `root`, which then returned the exit reason as its result.

Figure 2.2 shows an alternative trace, in which `PidB` dies *before* `root` creates a link to it, which generates an exit message with a different exit reason. The existence of these two different traces indicates a race condition when using `spawn` and `link` separately (which is the reason for the existence of an atomic `spawn_link` function in Erlang).

The diagrams help us to understand traces by gathering together all the events that affect one process into one box; in the original traces, these events may be scattered throughout the entire trace. But notice that the diagrams also *abstract away* from irrelevant information—specifically, the order in which messages are delivered to *different* processes, which is insignificant in Erlang. This abstraction is one strong reason why the diagrams are easier to understand than the traces they are generated from.

However, we *do* need to know the order in which calls to functions with side-effects occur, even if they are made in different processes. To make this order visible, we add dotted black arrows to our diagrams, from one side-effecting call to the next. Figure 2.3 illustrates one possible execution of this program, in which two processes race to write to the same file:

```
write_race() ->
  Pid = spawn(fun() ->
                file:write_file("a.txt","a")
              end),
  file:write_file("a.txt","b").
```

In this diagram, we can see that the `write_file` in the `root` process preceded the one in the spawned process `write_race.Pid`.

If we draw these arrows between *every* side-effect and its successor, then our diagrams rapidly become very cluttered. However, it is only necessary to indicate the sequencing of side-effects explicitly *if their sequence is not already determined*. For each pair of successive side-effect transitions, we thus compute Lamport's "happens before" relation [Lamport, 1978] between them, and

[5] If messages are consumed from a process mailbox out-of-order, then we show the delivery of a message to the mailbox, and its later consumption, as separate transitions.

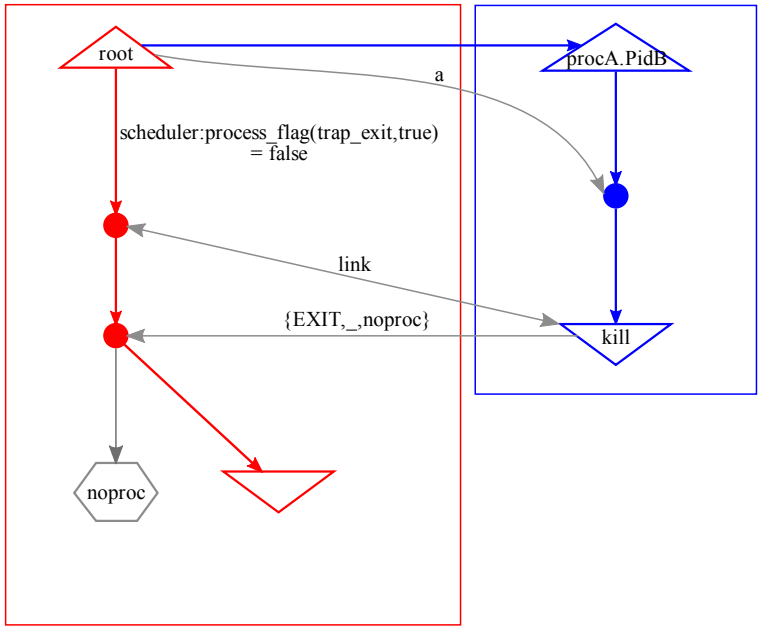

Figure 2.2: An alternative possible execution.

if this already implies that the first precedes the second, then we draw no arrow in the diagram. Interestingly, in our examples then this eliminates the majority of such arrows, and those that remain tend to surround possible race conditions—where the message passing (synchronization) does not enforce a particular order of side-effects. Thus black dotted arrows are often a sign of trouble.

6.1 Analyzing the `proc_reg` race conditions

Interestingly, as we saw in Section 5.5, when we instrumented `proc_reg` and tested it using PULSE and QuickCheck, we obtained a different—even simpler—minimal failing test case, than the one we had previously discovered using QuickCheck with the built-in Erlang scheduler. Since we need to use PULSE in order to obtain a trace to analyze, then we must fix this bug first, and see whether that also fixes the first problem we discovered. The failing test we find using PULSE is this one:

```
{[{set,{var,9},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,10},{call,proc_reg_eqc,kill,[{var,9}]}}],
 {[{set,{var,15},{call,proc_reg_eqc,reg,[c,{var,9}]}}],
  [{set,{var,12},{call,proc_reg_eqc,reg,[c,{var,9}]}}]}}
```

In this test case, we simply create a dead process (by spawning a process and then immediately killing it), and try to register it twice in parallel, and as it

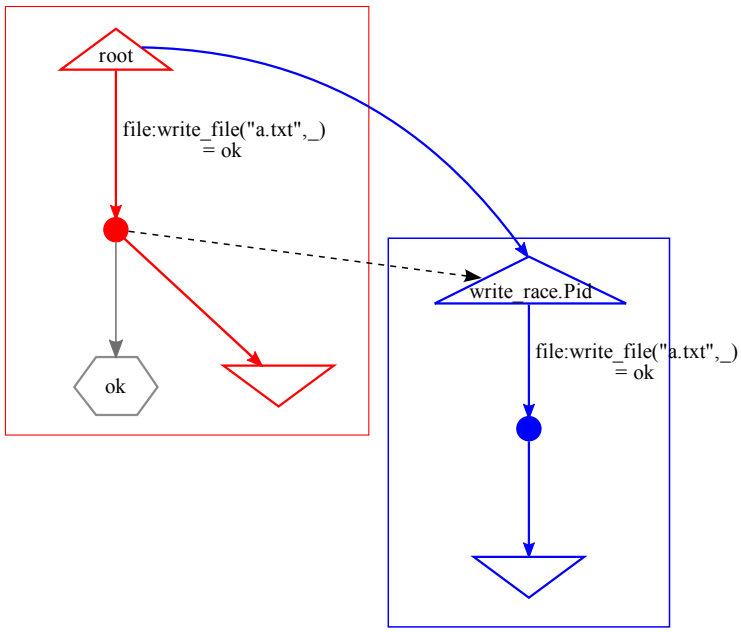

Figure 2.3: A race between two side-effects.

happens the first call to `reg` raises an exception. The diagram we generate is too large to include in full, but in Figure 2.4 we reproduce the part showing the problem.

In this diagram fragment, the processes we see are, from left to right, the `proc_reg` server, the second parallel fork (`BPid`), and the first parallel fork (`APid`). We can see that `BPid` first inserted its argument into the ETS table, recording that the name `c` is now taken, then sent an asynchronous message to the server (`{cast,{..}}`) to inform it of the new entry. Thereafter `APid` tried to insert an ETS entry with the same name—but failed. After discovering that the process being registered is actually dead, `APid` sent a message to the server asking it to "audit" its entry (`{call,{..},_,_}`)—that is, clean up the table by deleting the entry for a dead process. *But this message was delivered before the message from* `BPid`*!* As a result, the server could not find the dead process in its table, and failed to delete the entry created by `BPid`, leading `APid`'s second attempt to create an ETS entry to fail also—which is not expected to happen. When `BPid`'s message is finally received and processed by the server, it is already too late.

The problem arises because, while the clients create "forward" ETS entries linking the registered name to a pid, it is the server which creates a "reverse" entry linking the pid to its monitoring reference (created by the server). It is this reverse entry that is used by the server when asked to remove a dead

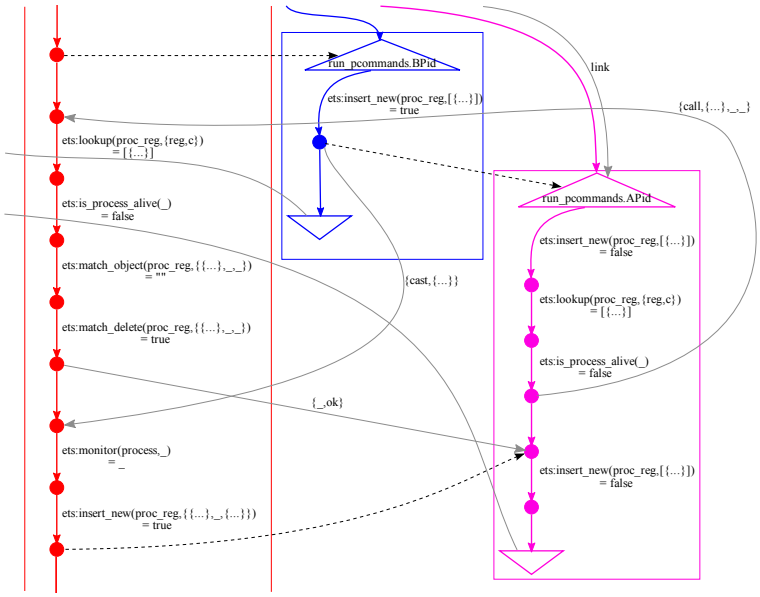

Figure 2.4: A problem caused by message overtaking.

process from its tables. We corrected the bug by letting clients (atomically) insert *two* ETS entries into the same table: the usual forward entry, and a dummy reverse entry (lacking a monitoring reference) that is later overwritten by the server. This dummy reverse entry enables the server to find and delete both entries in the test case above, thus solving the problem.

In fact, the current Erlang virtual machine happens to deliver messages to local mailboxes instantaneously, which means that one message cannot actually overtake another message sent earlier—the cause of the problem in this case. This is why this minimal failing test was not discovered when we ran tests on a multi-core, using the built-in scheduler. However, this behavior is not guaranteed by the language definition, and indeed, messages between nodes in a distributed system *can* overtake each other in this way. It is expected that future versions of the virtual machine may allow message overtaking even on a single "many-core" processor; thus we consider it an advantage that our scheduler allows this behavior, and can provoke race conditions that it causes.

It should be noted that exactly the same scenario can be triggered in an alternative way (without parallel processes and multi-core!); namely if the `BPid` above is preempted between its call to `ets:insert_new` and sending the `cast`-message. However, the likelihood for this is almost negligible, since the Erlang scheduler prefers running processes for relatively long time-slices. Using PULSE does not help triggering the scenario in this way either. PULSE is not in control at any point between `ets:insert_new` and sending the `cast`-message, meaning that only the Erlang scheduler controls the execution. Therefore, the only feasible way to repeatedly trigger this faulty scenario is by delaying the `cast`-message

by using PULSE (or a similar tool).

6.2 A second race condition in `proc_reg`

Having corrected the bug in `proc_reg` we repeated the QuickCheck test. The property still fails, with the same minimal failing case that we first discovered (which is not so surprising since the problem that we fixed in the previous section cannot actually occur with today's VM). However, we were now able to reproduce the failure with PULSE, as well as the built-in scheduler. As a result, we could now analyze and debug the race condition. The failing case is:

```
{[{set,{var,4},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,7},{call,proc_reg_eqc,kill,[{var,4}]}},
  {set,{var,12},{call,proc_reg_eqc,reg,[b,{var,4}]}}],
 {[{set,{var,18},{call,proc_reg_eqc,reg,[b,{var,4}]}}],
  [{set,{var,21},{call,proc_reg_eqc,reg,[b,{var,4}]}}]}}
```

In this test case we also create a dead process, but we try to register it once in the sequential prefix, before trying to register it twice in parallel. Once again, one of the calls to `reg` in the parallel branches raised an exception.

Turning again to the generated diagram, which is not included in the paper for space reasons, we observed that both parallel branches (`APid` and `BPid`) fail to insert `b` into the ETS table. They fail since the name `b` was already registered in the sequential part of the test case, and the server has not yet processed the `DOWN` message generated by the monitor. Both processes then call `where(b)` to see if `b` is *really* registered, which returns `undefined` since the process is dead. Both `APid` and `BPid` then request an "audit" by the server, to clean out the dead process. After the audit, both processes assume that it is now ok to register `b`, there is a race condition between the two processes, and one of the registrations fails. Since this is not expected, an exception is raised. (Note that if `b` were alive then this would be a perfectly valid race condition, where one of the two processes successfully registers the name and the other fails, but the specification says that the registration should always return `true` for dead processes).

This far into our analysis of the error it became clear that it is an altogether rather unwise idea ever to insert a dead process into the process registry. To fix the error we added a simple check that the process is alive before inserting it into the registry. The effect of this change on the performance turned out to be negligible, because `is_process_alive` is very efficient for local processes. After this change the module passed 20 000 tests, and we were satisfied.

7 Discussion and Related Work

Actually, the "fix" just described does not really remove all possible race conditions. Since the diagrams made us understand the algorithm much better, we can spot another possible race condition: If `APid` and `BPid` try to register the same pid at the same time, and that process dies *just after* `APid` and `BPid` have checked that it is alive, then the same problem we have just fixed, will arise. The reason that our tests succeeded even so, is that a test must contain *three* parallel branches to provoke the race condition in its new form—two processes

		K			
		2	3	4	5
	1	2	6	24	**120**
	2	6	90	**2520**	113400
	3	20	**1680**	369600	10^8
N	4	70	34650	6×10^7	3×10^{11}
	5	252	756756	10^{10}	6×10^{14}
	...		...		
	8	**12870**	10^{10}	10^{17}	8×10^{24}

Figure 2.5: Possible interleavings of parallel branches

making simultaneous attempts to register, and a third process to kill the pid concerned at the right moment. Because our parallel test cases only run *two* concurrent branches, then they can never provoke this behavior.

The best way to fix the last race condition problem in `proc_reg` would seem to be to simplify its API, by restricting `reg` so that a process may only register itself. This, at a stroke, eliminates the risk of two processes trying to register the same process at the same time, and guarantees that we can never try to register a dead process. This simplification was actually made in the production version of the code.

Parallelism in test cases

We could, of course, generate test cases with three, four, or even more concurrent branches, to test for this kind of race condition too. The problem is, as we explained in section 4.2, that the number of possible interleavings grows extremely fast with the number of parallel branches. The number of interleavings of K sequences of length N are as presented in Figure 2.5.

The practical consequence is that, if we allow more parallel branches in test cases, then we must restrict the length of each branch correspondingly. The bold entries in the table show the last "feasible" entry in each column—with three parallel branches, we would need to restrict each branch to just three commands; with four branches, we could only allow two; with five or more branches, we could allow only one command per branch. This is in itself a restriction that will make some race conditions impossible to detect. Moreover, with more parallel branches, there will be even more possible schedules for PULSE to explore, so race conditions depending on a precise schedule will be correspondingly harder to find.

There is thus an engineering trade-off to be made here: allowing greater parallelism in test cases may in theory allow more race conditions to be discovered, but in practice may reduce the probability of finding a bug with each test, while at the same time increasing the cost of each test. We decided to prioritize longer sequences over more parallelism in the test case, and so we chose $K = 2$. However, in the future we plan to experiment with letting QuickCheck randomly choose K and N from the set of feasible combinations. To be clear, note that K only refers to the parallelism in the test case, that is, the number

of processes that make calls to the API. The system under test may have hundreds of processes running, many of them controlled by PULSE, independently of K.

The problem of detecting race conditions is well studied and can be divided in runtime detection, also referred to as *dynamic detection*, and analyzing the source code, so called *static detection*. Most results refer to race conditions in which two threads or processes write to shared memory (data race condition), which in Erlang cannot happen. For us, a race condition appears if there are two schedules of occurring side effects (sending a message, writing to a file, trapping exits, linking to a process, etc) such that in one schedule our model of the system is violated and in the other schedule it is not. Of course, writing to a shared ETS table and writing in shared memory is related, but in our example it is allowed that two processes call ETS insert in parallel. By the atomicity of insert, one will succeed, the other will fail. Thus, there is a valid race condition that we do not want to detect, since it does not lead to a failure. Even in this slightly different setting, known results on race conditions still indicate that we are dealing with a hard problem. For example, Netzer and Miller [1990] show for a number of relations on traces of events that ordering these events on 'could have been a valid execution' is an NP-hard problem (for a shared memory model). Klein et al. [2003] show that statically detecting race conditions is NP-complete if more than one semaphore is used.

Thus, restricting `eqc_par_statem` to execute only two processes in parallel is a pragmatic choice. Three processes may be feasible, but real scalability is not in sight. This pragmatic choice is also supported by recent studies [Lu et al., 2008], where it is concluded that: *"Almost all (96%) of the examined concurrency bugs are guaranteed to manifest if certain partial order between 2 threads is enforced."*

Hierarchical approach

Note that our tools support a *hierarchical* approach to testing larger systems. We test `proc_reg` *under the assumption that the underlying ets operations are atomic*; PULSE does not attempt to (indeed, cannot) interleave the executions of single ETS operations, which are implemented by C code in the virtual machine. Once we have established that the `proc_reg` operations behave atomically, then *we can make the same assumption* about them when testing code that makes use of them. When testing for race conditions in modules that use `proc_reg`, then we need not, and do not want to, test for race conditions in `proc_reg` itself. As a result, the PULSE schedules remain short, and the simple random scheduling that we use suffices to find schedules that cause failures.

Model Checking

One could argue that the optimal solution to finding race conditions problem would be to use a model checker to explore all possible interleavings. The usual objections are nevertheless valid, and the rapidly growing state space for concurrent systems makes model checking totally infeasible, even with a model checker optimized for Erlang programs, such as McErlang [Fredlund

and Svensson, 2007]. Further it is not obvious what would be the property to model check, since the atomicity violations that we search for can not be directly translated into an LTL model checking property.

Input non-determinism

PULSE provides deterministic scheduling. However, in order for tests to be repeatable we also need the external functions to behave consistently across repeated runs. While marking them as side-effects will ensure that they are only called serially, PULSE does nothing to guarantee that functions called in the same sequence will return the same values in different runs. The user still has to make sure that the state of the system is reset properly before each run. Note that the same arguments apply to QuickCheck testing; it is crucial for shrinking and re-testing that input is deterministic and thus it works well to combine QuickCheck and PULSE.

False positives

In contrast to many race finding methods, that try to spot common patterns leading to concurrency bugs, our approach does not produce false positives and not even does it show races that result in correct execution of the program. This is because we employ *property-based testing* and classify test cases based on whether the results satisfy correctness properties and report a bug only when a property is violated.

Related tools

Park and Sen [2008] study atomicity in Java. Their approach is similar to ours in that they use a random scheduler both for repeatability and increased probability of finding atomicity violations. However, since Java communication is done with shared objects and locks, the analysis is rather different.

It is quite surprising that our simple randomized scheduler—and even just running tests on a multi-core—coupled with repetition of tests to reduce non-determinism, should work so well for us. After all, this can only work if the probability of provoking the race condition in each test that contains one is reasonably high. In contrast, race conditions are often regarded as very *hard* to provoke because they occur so rarely. For example, Sen used very carefully constructed schedules to provoke race conditions in Java programs [Sen, 2008a]—so how can we reasonably expect to find them just by running the same test a few times on a multi-core?

We believe two factors make our simple approach to race detection feasible.

- Firstly, Erlang is not Java. While there *is* shared data in Erlang programs, there is much less of it than in a concurrent Java program. Thus there are many fewer potential race conditions, and a simpler approach suffices to find them.
- Secondly, we are searching for race conditions during *unit testing*, where

each test runs for a short time using only a relatively small amount of code. During such short runs, there is a fair chance of provoking race conditions with any schedule. Finding race conditions during whole-program testing is a much harder problem.

Chess, developed by Musuvathi et al. [2008], is a system that shares many similarities with PULSE. Its main component is a scheduler capable of running the program deterministically and replaying schedules. The key difference between Chess and PULSE is that the former attempts to do an exhaustive search and enumerate all the possible schedules instead of randomly probing them. Several interesting techniques are employed, including prioritizing schedules that are more likely to trigger bugs, making sure that only fair schedules are enumerated and avoiding exercising schedules that differ insignificantly from already visited ones.

Visualization

Visualization is a common technique used to aid understanding software. Information is extracted statically from source code or dynamically from execution and displayed in graphical form. Of many software visualization tools a number are related to our work. Topol et al. [1995] developed a tool that visualizes executions of parallel programs and shows, among other things, a trace of messages sent between processes indicating the happened-before relationships. Work of Jerding et al. [1997] is able to show dynamic call-graphs of object-oriented programs and interaction patterns between their components. Arts and Fredlund [2002] describe a tool that visualizes traces of Erlang programs in form of abstract state transition diagrams. Artho et al. [2007] develop a notation that extends UML diagrams to also show traces of concurrent executions of threads, Maoz et al. [2007] create event sequence charts that can express which events "must happen" in all possible scenarios.

8 Conclusions

Concurrent code is hard to debug and therefore hard to get correct. In this paper we present an extension to QuickCheck, a user level scheduler for Erlang (PULSE), and a tool for visualizing concurrent executions that together help in debugging concurrent programs. The tools allow us to find concurrency errors on a module testing level, whereas industrial experience is that most of them slip through to system level testing, because the standard scheduler is deterministic, but behaves differently in different timing contexts.

We contributed `eqc_par_statem`, an extension of the state machine library for QuickCheck that enables parallel execution of a sequence of commands. We generate a sequential prefix to bring the system into a certain state and continue with parallel execution of a suffix of independent commands. As a result we can provoke concurrency errors and at the same time get good shrinking behavior from the test cases.

We contributed with PULSE, a user level scheduler that enables scheduling of any concurrent Erlang program in such a way that an execution can be repeated

deterministically. By randomly choosing different schedules, we are able to explore more execution paths than without such a scheduler. In combination with QuickCheck we get in addition an even better shrinking behavior, because of the repeatability of test cases.

We contributed with a graph visualization method and tool that enabled us to analyze concurrency faults more easily than when we had to stare at the produced traces. The visualization tool depends on the output produced by PULSE, but the use of computing the "happens before" relation to simplify the graph is a general principle.

We evaluated the tools on a real industrial case study and we detected two race conditions. The first one by only using `eqc_par_statem`; the fault had been noticed before, but now we did not need to instrument the code under test with `yield()` commands. The first and second race condition could easily be provoked by using PULSE. The traces recorded by PULSE were visualized and helped us in clearly identifying the sources of the two race conditions. By analyzing the graphs we could even identify a third possible race condition, which we could provoke if we allowed three instead of two parallel processes in `eqc_par_statem`.

Our contributions help Erlang software developers to get their concurrent code right and enables them to ship technologically more advanced solutions. Products that otherwise might have remained a prototype, because they were neither fully understood nor tested enough, can now make it into production. The tool PULSE and the visualization tool are available under the Simplified BSD License and have a commercially supported version as part of Quviq QuickCheck.

Acknowledgements

This research was sponsored by EU FP7 Collaborative project *ProTest*, grant number 215868.

Paper III

Ranking Programs using Black Box Testing

This paper was originally presented at the AST 2010 workshop in Cape Town. This is an extended version which has been submitted to a special issue of the Software Quality Journal.

Ranking Programs using Black Box Testing

Koen Claessen, John Hughes, Michał Pałka, Nick Smallbone, Hans Svensson

Abstract

We present an unbiased method for measuring the relative quality of different solutions to a programming problem. Our method is based on identifying possible bugs from program behaviour through black-box testing. The main motivation for such a method is its use in experimental evaluation of software development methods. We report on the use of our method in a small-scale such experiment, which was aimed at evaluating the effectiveness of property-based testing vs. unit testing in software development. The experiment validated the assessment method and yielded suggestive, though not yet statistically significant results. We also show tests that justify our method.

1 Introduction

Property-based testing is an approach to testing software against a formal specification, consisting of universally quantified *properties* which supply both test data generators and test oracles. QuickCheck is a property-based testing tool first developed for Haskell [Claessen and Hughes, 2000], and which forms the basis for a commercial tool developed by Quviq [Arts et al., 2006]. As a simple example, using QuickCheck, a programmer could specify that list reversal is its own inverse like this,

```
prop_reverse (xs :: [Integer]) =
  reverse (reverse xs) == xs
```

which defines a property called `prop_reverse` which is universally quantified over all lists of integers `xs`. Given such a property, QuickCheck generates random values for `xs` as test data, and uses the body of the property as an oracle to decide whether each test has passed. When a test fails, QuickCheck *shrinks* the failing test case, searching systematically for a minimal failing example, in a way similar to delta-debugging [Zeller, 2002]. The resulting minimal failing case usually makes diagnosing a fault easy. For example, if the programmer erroneously wrote

```
prop_reverse (xs :: [Integer]) =
  reverse xs == xs
```

then QuickCheck would report the minimal counterexample `[0,1]`, since at least two different elements are needed to violate the property, and the two smallest different integers are `0` and `1`.

The idea of testing code against general properties, rather than specific test cases, is an appealing one which also underlies Tillmann and Schulte's *parameterized unit tests* [Tillmann and Schulte, 2005] and the Pex tool [Tillmann and de Halleux, 2008] (although the test case generation works differently). We believe firmly that it brings a multitude of benefits to the developer, improving quality and speeding development by revealing problems faster and earlier. Yet claims such as this are easy to make, but hard to prove. And it is not *obvious* that property-based testing must be superior to traditional test automation. Among the possible *disadvantages* of QuickCheck testing are:

- it is often necessary to write *test data generators* for problem-specific data structures, code which is not needed at all in traditional testing.
- the developer must *formulate a formal specification*, which is conceptually more difficult than just predicting the correct output in specific examples.
- randomly generated test cases might potentially be less effective at revealing errors than carefully chosen ones.

Thus an empirical comparison of property-based testing against other methods is warranted.

Our overall goal is to evaluate property-based testing as a development tool, by comparing programs developed by students using QuickCheck for testing,

against programs developed for the same problem using HUnit [Herington, 2010]—a unit testing framework for Haskell similar to the popular JUnit tool for Java programmers [JUnit.org, 2010]. We have not reached this goal yet—we have carried out a small-scale experiment, but we need more participants to draw statistically significant conclusions. However, we have identified an important problem to solve along the way: how should we *rank* student solutions against each other, without introducing experimental bias?

Our intention is to rank solutions by testing them: those that pass the most tests will be ranked the highest. But the choice of *test suite* is critical. It is tempting to use QuickCheck to test student solutions against our own properties, using the proportion of tests passed as a measure of quality—but this would risk experimental bias in two different ways:

- By using one of the tools in the comparison to grade solutions, we might unfairly bias the experiment to favour that tool.
- The ranking of solutions could depend critically on the distribution of random tests, which is rather arbitrary.

Unfortunately, a manually constructed set of test cases could also introduce experimental bias. If we were to include many similar tests of a particular kind, for example, then handling that kind of test successfully would carry more weight in our assessment of solutions than handling other kinds of test.

Our goal in this paper, thus, is to develop a way of ranking student solutions by testing that leaves no room for the experimenter's bias to affect the result. We will do so by generating a set of test cases *from the submissions themselves*, based on a simple "bug model" presented in Section 3, such that each test case tests for one bug. We then rank solutions by the number of bugs they contain. QuickCheck is used to help find this set of test cases, but in such a way that the distribution of random tests is of almost no importance.

1.0.1 Contribution

The main contribution of this paper is the ranking method we developed. As evidence that the ranking is reasonable, we present the results of our small-scale experiment, in which solutions to three different problems are compared in this way. We also evaluate the stability of the method further by ranking *mutated* versions of the same program.

The remainder of the paper is structured as follows. In the next section we briefly describe the experiment we carried out. In Section 3 we explain and motivate our ranking method. Section 4 analyses the results obtained, and Section 5 checks that our ranking behaves reasonably. In Section 6 we discuss related work, and we conclude in Section 7.

2 The Experiment

We designed an experiment to test the hypothesis that *"Property-based testing is more effective than unit testing, as a tool during software development"*,

using QuickCheck as the property-based testing tool, and HUnit as the unit testing tool. We used a *replicated project study* [Basili et al., 1986], where in a controlled experiment a group of student participants individually solved three different programming tasks. We planned the experiment in accordance to best practice for such experiments; trying not to exclude participants, assigning the participants randomly to tools, using a variety of programming tasks, and trying our best not to influence the outcome unnecessarily. We are only evaluating the final product, thus we are not interested in process aspects in this study.

In the rest of this section we describe in more detail how we planned and executed the experiment. We also motivate the choice of programming assignments given to the participants.

2.1 Experiment overview

We planned an experiment to be conducted during one day. Since we expected participants to be unfamiliar with at least one of the tools in the comparison, we devoted the morning to a training session in which the tools were introduced to the participants. The main issue in the design of the experiment was the programming task (or tasks) to be given to the participants. Using several different tasks would yield more data points, while using one single (bigger) task would give us data points of higher quality. We decided to give three separate tasks to the participants, mostly because by doing this, and selecting three different types of problems, we could reduce the risk of choosing a task particularly suited to one tool or the other. All tasks were rather small, and require only 20–50 lines of Haskell code to implement correctly.

To maximize the number of data points we decided to assign the tasks to individuals instead of forming groups. Repeating the experiments as a pair-programming assignment would also be interesting.

2.2 Programming assignments

We constructed three separate programming assignments. We tried to choose problems from three different categories: one data-structure implementation problem, one search/algorithmic problem, and one slightly tedious string manipulation task.

2.2.1 Problem 1: email anonymizer

In this task the participants were asked to write a sanitizing function `anonymize` which blanks out email addresses in a string. For example,

```
anonymize "pelle@foretag.se" ==
  "p____@f______.s_"

anonymize "Hi johnny.cash@music.org!" ==
  "Hi j_____.c___@m____.o__!"
```

The function should identify all email addresses in the input, change them, but leave all other text untouched. This is a simple problem, but with a lot of tedious cases.

2.2.2 Problem 2: Interval sets

In this task the participants were asked to implement a compact representation of sets of integers based on lists of intervals, represented by the type `IntervalSet = [(Int,Int)]`, where for example the set `{1,2,3,7,8,9,10}` would be represented by the list `[(1,3),(7,10)]`. The participants were instructed to implement a family of functions for this data type (`empty`, `member`, `insert`, `delete`, `merge`). There are many special cases to consider—for example, inserting an element between two intervals may cause them to merge into one.

2.2.3 Problem 3: Cryptarithm

In this task the students were asked to write a program that solves puzzles like this one:

```
 SEND
 MORE
-----+
MONEY
```

The task is to assign a mapping from letters to (unique) digits, such that the calculation makes sense. (In the example `M = 1, O = 0, S = 9, R = 8, E = 5, N = 6, Y = 2, D = 7`). Solving the puzzle is complicated by the fact that there might be more than one solution and that there are problems for which there is no solution. This is a search problem, which requires an algorithm with some level of sophistication to be computationally feasible.

2.3 The participants

Since the university (Chalmers University of Technology, Gothenburg, Sweden) teaches Haskell, this was the language we used in the experiment. We tried to recruit students with (at least) a fair understanding of functional programming. This we did because we believed that too inexperienced programmers would not be able to benefit from either QuickCheck or HUnit. The participants were recruited by advertising on campus, email messages sent to students from the previous Haskell course and announcements in different ongoing courses. Unfortunately the only available date collided with exams at the university, which lowered the number of potential participants. In the end we got only 13 participants. This is too few to draw statistically significant conclusions, but on the other hand it is a rather manageable number of submissions to analyze in greater detail. Most of the participants were at a level where they had passed (often with honour) a 10-week programming course in Haskell.

2.4 Assigning the participants into groups

We assigned the participants randomly (by lot) into two groups, one group using QuickCheck and one group using HUnit.

2.5 Training the participants

The experiment started with a training session for the participants. The training was divided into two parts, one joint session, and one session for the specific tool. In the first session, we explained the purpose and the underlying hypothesis for the experiment. We also clearly explained that we were interested in software quality rather than development time. The participants were encouraged to use all of the allocated time to produce the best software possible.

In the second session the groups were introduced to their respective testing tools, by a lecture and practical session. Both sessions lasted around 60 minutes.

2.6 Programming environment

Finally, with everything set up, the participants were given the three different tasks with a time limit of 50 minutes for each of the tasks. The participants were each given a computer equipped with GHC (the Haskell compiler) [The GHC Team, 2010], both the testing tools, and documentation. The computers were connected to the Internet, but since the participants were aware of the purpose of the study and encouraged not to use other tools than the assigned testing tool it is our belief this did not affect the outcome of the experiment.[1]

2.7 Data collection and reduction

Before the experiment started we asked all participants to fill out a survey, where they had to indicate their training level (programming courses taken) and their estimated skill level (on a 1–5 scale).

From the experiments we collected the implementations as well as the testing code written by each participant.

2.7.1 Manual grading of implementations

Each of the three tasks were graded by an experienced Haskell programmer. We graded each implementation on a 0–10 scale, just as we would have graded an exam question. Since the tasks were reasonably small, and the number of participants manageable, this was feasible. To prevent any possible bias, the grader was not allowed to see the testing code and thus he could not know whether each student was using QuickCheck or HUnit.

2.7.2 Automatic ranking

The implementations of each problem were subjected to an analysis that we present in Section 3.

We had several students submit uncompilable code.[2] In those cases, we made

[1] Why not simply disconnect the computers from the Internet? Because we used an online submission system, as well as documentation and software from network file systems.

[2] Since we asked students to submit their code at a fixed time, some students submitted in the middle of making changes.

the code compile by for example removing any ill-formed program fragments. This was because such a program might be partly working, and deserve a reasonable score; we thought it would be unfair if it got a score of zero simply because it (say) had a syntax error.

2.7.3 Grading of test suites

We also graded participants' testing code. Each submission was graded by hand by judging the completeness of the test suite—and penalised for missing cases (for HUnit) or incomplete specifications (for QuickCheck). As we did not instruct the students to use test-driven development, there was no penalty for not testing a function if that function was not implemented.

2.7.4 Cross-comparison of tests

We naturally wanted to automatically grade students' test code too—not least, because a human grader may be biased towards QuickCheck or HUnit tests. Our approach was simply to take each student's test suite, and run it against all of the submissions we had; for every submission the test suite found a bug in, it scored one point.

We applied this method successfully to the interval sets problem. However, for the anonymizer and cryptarithm problems, many students performed white box testing, testing functions that were internal to their implementation; therefore we were not able to transfer test suites from one implementation to another, and we had to abandon the idea for these problems.

3 Evaluation Method

We assume we have a number of student *answers* to evaluate, $A_1, \ldots, A_n$, and a perfect solution A_0, each answer being a program mapping a test case to output. We assume that we have a test oracle which can determine whether or not the output produced by an answer is correct, for any possible test case. Such an oracle can be expressed as a QuickCheck property—if the correct output is unique, then it is enough to compare with A_0's output; otherwise, something more complex is required. Raising an exception, or falling into a loop,[3] is never correct behaviour. We can thus determine, for an arbitrary test case, which of the student answers pass the test.

We recall that the purpose of our automatic evaluation method is to find a set of test cases that is as unbiased as possible. In particular, we want to avoid counting multiple test cases that are equivalent, in the sense that they trigger the same bug.

Thus, we aim to "count the bugs" in each answer, *using black-box testing alone*. How, then, should we define a "bug"? We cannot refer to errors at specific places in the source code, since we use black-box testing only—we must define a "bug" in terms of the program behaviour. We take the following as our bug model:

[3] Detecting a looping program is approximated by an appropriately chosen timeout.

- A *bug* causes a program to fail for a set of test cases. Given a bug b, we write the set of test cases that it causes to fail as $BugTests(b)$. (Note that it is possible that the same bug b occurs in several different programs.)
- A *program* p will contain a set of bugs, $Bugs(p)$. The set of test cases that p fails for will be

$$FailingTests(p) = \bigcup_{b \in Bugs(p)} BugTests(b)$$

It is quite possible, of course, that two different errors in the source code might manifest themselves in the same way, causing the same set of tests to fail. We will treat these as the *same* bug, quite simply because there is no way to distinguish them using black-box testing.

It is also possible that two different bugs in combination might "cancel each other out" in some test cases, leading a program containing both bugs to behave correctly, despite their presence. We cannot take this possibility into account, once again because black-box testing cannot distinguish correct output produced "by accident" from correct output produced correctly. We believe the phenomenon, though familiar to developers, is rare enough not to influence our results strongly.

Our approach is to analyze the failures of the student answers, and use them to infer the existence of possible bugs $Bugs$, and their failure sets. Then we shall rank each answer program A_i by the number of these bugs that the answer appears to contain:

$$rank(A_i) = |\{b \in Bugs \mid BugTests(b) \subseteq FailingTests(A_i)\}|$$

In general, there are many ways of explaining program failures via a set of bugs. The most trivial is to take each answer's failure set $FailingTests(A_i)$ to represent a different possible bug; then the rank of each answer would be the number of other (different) answers that fail on a strictly smaller set of inputs. However, we reject this idea as too crude, because it gives no insight into the *nature* of the bugs present. We shall aim instead to find a more refined set of possible bugs, in which each bug explains a small set of "similar" failures.

Now, let us define the failures of a *test case* to be the set of answers that it provokes to fail:

$$AnswersFailing(t) = \{A_i \mid t \in FailingTests(A_i)\}$$

We insist that if two test cases t_1 and t_2 provoke the same answers to fail, then they are equivalent with respect to the bugs we infer:

$$AnswersFailing(t_1) = AnswersFailing(t_2) \Longrightarrow \\ \forall b \in Bugs.\ t_1 \in BugTests(b) \Leftrightarrow t_2 \in BugTests(b)$$

We will not distinguish such a pair of test cases, because there is no evidence from the answers that could justify doing so. Thus we can partition the space of test cases into subsets that behave equivalently with respect to our answers. By identifying bugs with these partitions (except, if it exists, the partition

which causes no answers to fail), then we obtain a maximal set of bugs that can explain the failures we observe. No other set of bugs can be more refined than this without distinguishing inputs that should not be distinguished.

However, we regard this partition as a little *too* refined. Consider two answers A_1 and A_2, and three partitions B, B_1 and B_2, such that

$$\begin{aligned}&\forall t \in B.\ \mathit{AnswersFailing}(t) = \{A_1, A_2\}\\ &\forall t \in B_1.\ \mathit{AnswersFailing}(t) = \{A_1\}\\ &\forall t \in B_2.\ \mathit{AnswersFailing}(t) = \{A_2\}\end{aligned}$$

Clearly, one possibility is that there are three separate bugs represented here, and that

$$\begin{aligned}&\mathit{Bugs}(A_1) = \{B, B_1\}\\ &\mathit{Bugs}(A_2) = \{B, B_2\}\end{aligned}$$

But another possibility is that there are only *two* different bugs represented, $B_1' = B \cup B_1$ and $B_2' = B \cup B_2$, and that each A_i just has one bug, B_i'. In this case, test cases in B can provoke either bug. Since test cases which can provoke several different bugs are quite familiar, then we regard the latter possibility as more plausible than the former. We choose therefore to *ignore any partitions whose failing answers are the union of those of a set of other partitions*; we call these partitions *redundant*, and we consider it likely that the test cases they contain simply provoke several bugs at once. In terms of our bug model, we combine such partitions with those representing the individual bugs whose union explains their failures. Note, however, that if a *third* answer A_3 only fails for inputs in B, then we consider this evidence that B does indeed represent an independent bug (since $\{A_1, A_2, A_3\}$ is *not* the union of $\{A_1\}$ and $\{A_2\}$), and that answers A_1 and A_2 therefore contain two bugs each.

Now, to rank our answers we construct a test suite containing one test case from each of the remaining partitions, count the tests that each answer fails, and assign ranks accordingly.

In practice, we find the partitions by running a very large number of random tests. We maintain a set of test cases *Suite*, each in a different partition. For each newly generated test case t, we test all of the answers to compute $\mathit{AnswersFailing}(t)$. We then test whether the testcase is redundant in the sense described above:

$$\begin{aligned}&\mathit{Redundant}(t, \mathit{Suite}) \mathrel{\hat{=}}\\ &\quad \mathit{AnswersFailing}(t) =\\ &\qquad \bigcup \left\{ \mathit{AnswersFailing}(t') \;\middle|\; \begin{array}{l} t' \in \mathit{Suite},\\ \mathit{AnswersFailing}(t') \subseteq\\ \quad \mathit{AnswersFailing}(t) \end{array} \right\}\end{aligned}$$

Whenever t is not redundant, i.e. when $\mathit{Redundant}(t, \mathit{Suite})$ evaluates to *False*, then we apply QuickCheck's shrinking to find a *minimal* t_{min} that is not redundant with respect to *Suite*—which is always possible, since if we cannot find any smaller test case which is irredundant, then we can just take t itself. Then we *add* t_{min} to *Suite*, and *remove* any $t' \in \mathit{Suite}$ such that $\mathit{Redundant}(t', (\mathit{Suite} - t') \cup \{t_{min}\})$. (Shrinking at this point probably helps us to find test cases that provoke a single bug rather than several—"probably"

since a smaller test case is likely to provoke fewer bugs than a larger one, but of course there is no guarantee of this).

We continue this process until a large number of random tests fail to add any test cases to *Suite*. At this point, we assume that we have found one test case for each irredundant input partition, and we can use our test suite to rank answers.

Note that this method takes no account of the *sizes* of the partitions involved—we count a bug as a bug, whether it causes a failure for only one input value, or for infinitely many. Of course, the severity of bugs in practice may vary dramatically depending on precisely *which* inputs they cause failures for—but taking this into account would make our results dependent on value judgements about the importance of different kinds of input, and these value judgements would inevitably introduce experimental bias.

In the following section, we will see how this method performs in practice.

4 Analysis

We adopted the statistical *null hypothesis* to be that there is no difference in quality between programs developed using QuickCheck and programs developed using HUnit. The aim of our analysis will be to establish whether the samples we got are different in a way which cannot be explained by coincidence.

We collected solutions to all three tasks programmed by 13 students, 7 of which were assigned to the group using QuickCheck and the remaining 6 to one using HUnit. In this section we will refer to the answers (solutions to tasks) as $A1$ to $A13$. Since the submissions have been anonymized, numbering of answers have also been altered and answers $A1$ to different problems correspond to submissions of different participants. For each task there is also a special answer $A0$ which is the model answer which we use as the testing oracle. For the anonymizer, we also added the identity function for comparison as $A14$, and for the interval sets problem we added a completely undefined solution as $A14$.

4.1 Automatic Ranking of Solutions

We ranked all solutions according to the method outlined in Section 3. The ranking method produced a test-suite for each of the three tasks and assigned the number of failing tests to each answer of every task. The final score that we used for evaluation of answers was the *number of successful runs on tests from the test-suite*. The generated test suites are shown in Figure 3.1. Every test in the test suite causes some answer to fail; for example `delete 0 []` is the simplest test that causes answers that did not implement the `delete` function to fail. These test cases have been shrunk by QuickCheck, which is why the only letter to appear in the anonymizer test cases is 'a', and why the strings are so short[4].

[4]Because Haskell encourages the use of dynamic data-structures, then none of the solutions could encounter a buffer overflow or other error caused by fixed size arrays. As a result, there is no need for tests with very long strings.

Anon	IntSet	Crypt
""	member 0 []	b+b=c
"\n"	member 0 [(-2,2)]	a+a=a
"@"	member 2 [(1,1)]	a+b=ab
"a"	member 0 [(-3,-3),(0,4)]	aa+a=bac
"&@"	insert 0 []	
".@"	insert -1 [(1,1)]	
"@@"	insert 0 [(-2,0)]	
".@@"	insert 1 [(-2,0)]	
"@_a"	insert 2 [(0,0)]	
"@a="	delete 0 []	
"_ &"	delete 0 [(0,0)]	
"a@a"	delete 0 [(0,1)]	
"#@&@"	merge [] []	
".a@#"	merge [] [(-1,0)]	
"a@_a"	merge [(0,0)] [(0,0)]	
"a@aa"	merge [(-1,0),(2,3)] [(-1,0)]	

Figure 3.1: Generated test suites.

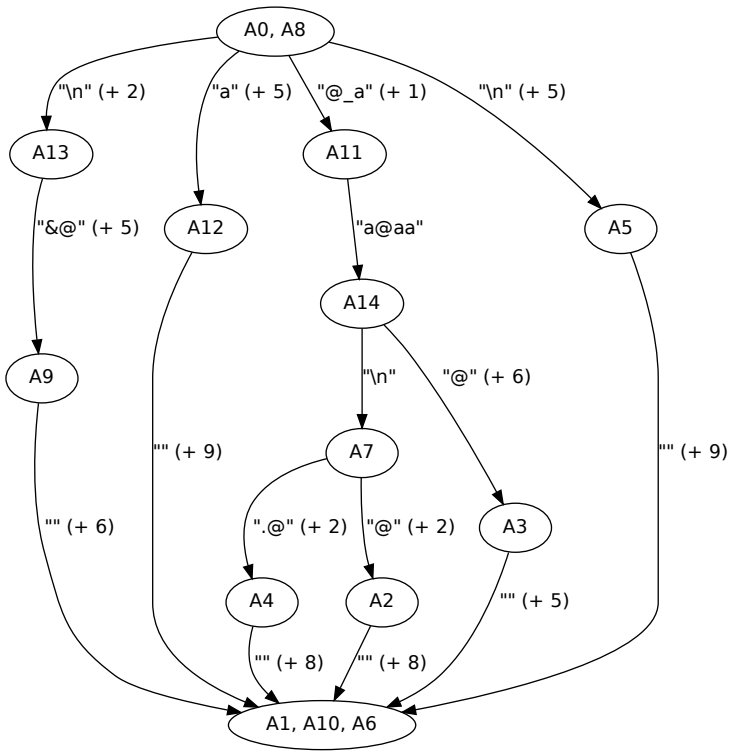

Figure 3.2: Relative correctness of anonymizer answers.

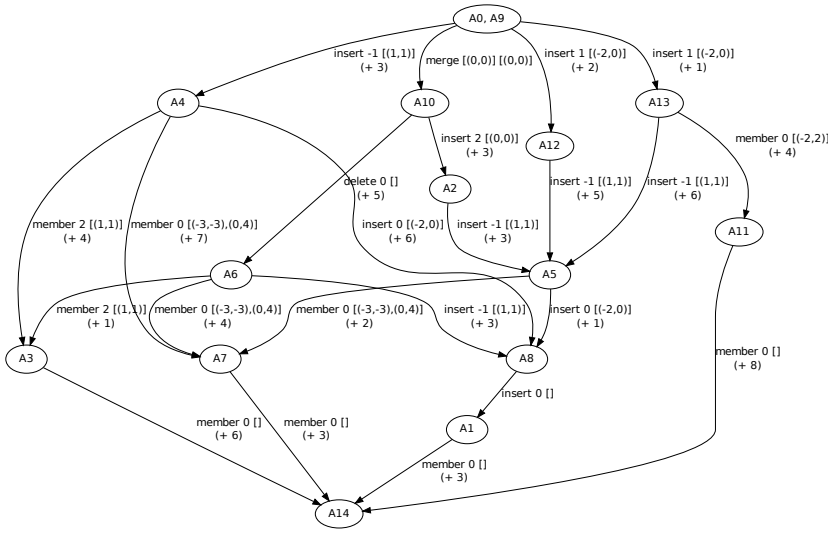

Figure 3.3: Relative correctness of interval set answers.

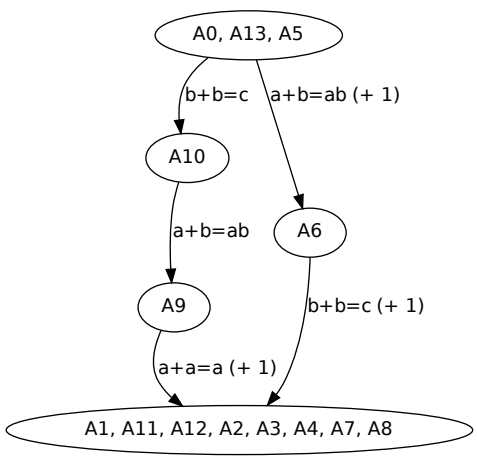

Figure 3.4: Relative correctness of cryptarithm answers.

Answer	Anon	IntSet	Crypto
A0	16	16	4
A1	0*	4*	0*
A2	9*	11*	0
A3	6	7*	0*
A4	9*	12*	0*
A5	10	7	4*
A6	0	9	2*
A7	12*	4	0*
A8	16*	5*	0
A9	7	16	2
A10	0	15*	3
A11	14	9	0*
A12	10*	13	0
A13	13*	14*	4

Figure 3.5: Results of automatic grading.

Figures 3.2 to 3.4 visualize the test results. Each node represents a set of answers which pass precisely the same tests. An arrow from one node to another means that the answers at the target of the arrow pass a subset of the tests that the answers at the source of the arrow pass. Arrows are labelled with a test case that distinguishes the source and target, and the number of other such test cases in brackets. For instance, we can read from Figure 3.2 that $A2$ fails three more tests than $A7$, and that it fails on the input string "@" whereas $A7$ succeeds on it. Thus these figures visualize a "correctness partial order" on the submitted answers.

The top node of each graph represents the entirely correct solutions, including the model answer A0. The bottom node represents incomplete solutions, in which the main functions were not defined—and which therefore fail all tests. Interestingly, our analysis *distinguishes all other answers*—no two partially correct submissions were equivalent. Moreover, there is a non-trivial partial ordering of answers in each case: some answers really are strictly better than others. We conclude that our analysis is able to classify partially correct answers in an interesting way. (We also conclude that the cryptarithm problem was too hard to solve in the time available, since more than half of the submissions failed every test).

The final score assigned to each answer is shown in figure 3.5. In order to assign better answers a higher score, we show the number of tests *passed* by each answer, rather than the number of test failures—i.e. bugs. $A0$ is the model answer in each case, and answers coming from the group assigned to using QuickCheck are marked with stars(*).

The following table shows a statistical analysis of scores from the automatic ranking. To determine whether there is a statistical difference between samples coming from the two groups we applied Welch's t-test (which tests whether two collections of data have the same mean) and got the values visible in the P-value row (which we shall explain below).

	Anon	IntSet	Crypto
All - Avg (Sdev)	8.15 (5.38)	9.69 (4.15)	1.15 (1.63)
QC - Avg (Sdev)	9.86 (5.01)	9.71 (4.39)	0.86 (1.57)
HU - Avg (Sdev)	6.17 (5.53)	9.67 (4.27)	1.50 (1.76)
P-value	0.2390	0.9846	0.5065

For the anonymizer example, we can see that the solutions developed using QuickCheck scored higher than those developed using HUnit, for interval sets the scores were about the same, and for the cryptarithm example, then solutions developed using QuickCheck fared worse. The P-value is the probability of seeing the observed (or lower) difference in scores by sheer chance, if there is no difference in the expected score using HUnit and QuickCheck (the null hypothesis). For the anonymizer problem then the null hypothesis can be rejected with a confidence of 76%—which is not enough to draw a statistically significant conclusion from, but is nevertheless suggestive enough to encourage us to perform a bigger experiment.

4.2 Manual Grading of Solutions

In the table below we present that average scores (and their standard deviations) from the manual grading for the three problems. These numbers are not conclusive from a statistical point of view. Thus, for the manual grading we can not reject the null hypothesis. Nevertheless, there is a tendency corresponding to the results of the automatic grading in section 4.1. For example, in the email anonymizer problem the solutions that use QuickCheck are graded higher than the solutions that use HUnit.

	Anon	IntSet	Crypto
All - Avg (Sdev)	4.07 (2.78)	4.46 (2.87)	2.15 (2.91)
QC - Avg (Sdev)	4.86 (2.67)	4.43 (2.88)	1.86 (3.23)
HU - Avg (Sdev)	3.17 (2.86)	4.50 (3.13)	2.50 (2.74)

To further justify our method for automatic ranking of the solutions, we would like to see a correlation between the automatic scores and the manual scores. However, we can not expect them to be exactly the same since the automatic grading is in a sense less forgiving. (The automatic grading measures how well the program actually works, while the manual grading measures "how far from a correct program" the solution is.) If we look in more detail on the scores to the email anonymizer problem, presented in the table below, we can see that although the scores are not identical, they tend to rank the solutions in a very similar way. The most striking difference is for solution $A7$, which is ranked 4th by the automatic ranking and 10th by the manual ranking. This is caused by the nature of the problem. The *identity function* (the function simply returning the input, $A14$) is actually a rather good approximation of the solution functionality-wise. $A7$ is close to the identity function—it does almost nothing, getting a decent score from the automatic grading, but failing to impress a human marker.

Answer	Auto	Manual	Auto rank	Manual rank
A1	0	3	11	8
A2	9	3	7	8
A3	6	2	10	10
A4	9	5	7	4
A5	10	4	5	5
A6	0	0	11	13
A7	12	2	4	10
A8	16	9	1	1
A9	7	4	9	5
A10	0	1	11	12
A11	14	8	2	2
A12	10	4	5	5
A13	13	8	3	2

4.3 Assessment of Students' Testing

As described in Section 2.7.3, we checked the quality of each student's test code both manually and automatically (by counting how many submissions each test suite could detect a bug in). Figure 3.6 shows the results.

	Student number						
QuickCheck	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Manual grading	0	0	0	3	9	9	12
Automatic grading	0	0	0	0	8	10	11

	Student number					
HUnit	**8**	**9**	**10**	**11**	**12**	**13**
Manual grading	3	12	6	3	6	9
Automatic grading	0	5	5	6	7	8

Figure 3.6: Manual vs. automatic grading of test suite quality.

The manual scores may be biased since all the authors are QuickCheck aficionados, so we would like to use them only as a "sanity check" to make sure that the automatic scores are reasonable. We can see that, broadly speaking, the manual and automatic scores agree.

The biggest discrepancy is that student 9 got full marks according to our manual grading but only 5/11 according to the automatic grading. The main reason is that his test suite was less comprehensive than we thought: it included several interesting edge cases, such as an insert that "fills the gap" between two intervals and causes them to become one larger interval, but left out some simple cases, such as `insert 2 (insert 0 empty)`. In this case, the automatic grader produced the fairer mark.

So, the automatically-produced scores look reasonable and we pay no more attention to the manual scores. Looking at the results, we see that four students from the QuickCheck group were not able to detect any bugs at all. (Three of

them submitted no test code at all[5], and one of them just tested one special case of the `member` function.) This compares to just one student from the HUnit group who was unable to find any bugs.

However, of the students who submitted a useful test suite, the *worst* QuickCheck test suite got the same score as the *best* HUnit test suite! All of the HUnit test suites, as it happens, were missing some edge case or other.[6]

So, of the students who were using QuickCheck, half failed to submit any useful test-suite at all, and the other half's test suites were the best ones submitted. There may be several explanations for this: perhaps QuickCheck properties are harder to write but more effective than unit tests; or perhaps QuickCheck is only effective in the hands of a strong programmer; or perhaps QuickCheck properties are "all-or-nothing", so that a property will either be ineffective or catch a wide range of bugs; or perhaps it was just a coincidence. The "all-or-nothing"-effect, or more often referred to as the "double hump"-effect, is often observed in introductory programming classes [Dehnadi and Bornat, 2006]. It would certainly be interesting to see if the same pattern applies to property-based testing; this is something we will aim to find out in our next experiment.

5 Correctness and Stability of the Bug Measure

To justify our ranking method, we checked the quality of the rankings achieved in three different ways:

- We evaluated the stability of the ranking of the actual student solutions.
- We used *program mutation* to create a larger sample of programs. Thereafter we measured how stable the relative ranking of a randomly chosen pair of programs is, when we alter the other programs that appear in the ranking.
- We tested how well the bugs found by the ranking algorithm correspond to the actual bugs in the programs.

We would ideally like to *prove* things about the algorithm, but there is no objective way to tell a "good" ranking from a "bad" ranking so no complete specification of the ranking algorithm. Furthermore, any statistical properties about stability are likely to be false in pathological cases. So we rely on tests instead to tell us if the algorithm is any good.

5.1 Stability with Respect to Choosing a Test Suite

Our bug-analysis performs a random search in the space of test cases in order to construct its test suite. Therefore, it is possible that different searches select different sets of tests, and thus assign different ranks to the same program in

[5] Of course, this does not imply that these students did not test their code *at all*—just that they did not automate their tests. Haskell provides a read-eval-print loop which makes interactive testing quite easy.

[6] Functions on interval sets have a surprising number of edge cases; with QuickCheck, there is no need to enumerate them.

different runs. To investigate this, we ran the bug analysis ten times on the solutions to each of the three problems. We found that the partial ordering on solutions that we inferred did not change, but the size of test suite did vary slightly. This could lead to the same solution failing a different number of tests in different runs, and thus to a different rank being assigned to it. The table below shows the results for each problem. Firstly, the number of consecutive tests we ran without refining the test suite before concluding it was stable. Secondly, the sizes of the test suites we obtained for each problem. Once a test suite was obtained, we assigned a rank to each answer, namely the number of tests it failed. These ranks did differ between runs, but the rank of each answer never varied by more than one in different runs. The last rows show the average and maximum standard deviations of the ranks assigned to each answer.

	Anon	IntSet	Crypto
Number of tests	10000	10000	1000
Sizes of test suite	15,16	15,16	4
Avg std dev of ranks	0.08	0.06	0
Max std dev of ranks	0.14	0.14	0

From this we conclude that the rank assignment is not much affected by the random choices made as we construct the test suite.

5.2 Stability of Score in Different Rankings

Our ranking method assigns a score to a program based not only on its failures but also on the failures of the other programs that participate in the ranking. Since the generated test suite depends on the exact set of programs that are ranked the same program will in general get different scores in two different rankings, even if we rule out the influence of test selection (as tested for in Section 5.1). Furthermore, the difference in scores can be arbitrarily large if we choose the programs maliciously or in other pathological cases.[7] We would like to formulate a stability criterion but we must test it in a relaxed form: instead of requiring absolute stability we will expect rankings to be stable when typical programs are ranked and not care what happens when we choose the programs maliciously.

Testing our method on "typical" programs requires having a supply of them. Of course we would like to possess a large library of programs written by human beings, but unfortunately we only have a small number of student submissions. Thus, we decided to employ the technique of *program mutation* to simulate having many reasonable programs. We started with the model solution for the email anonymiser problem and identified 18 *mutation points*, places in the program that we could modify to introduce a bug. These modifications are called *mutations*. Some mutation points we could modify in several ways to get different bugs, and some mutations excluded others. In this way we got a set of 3128 *mutants*, or buggy variations of the original program.

Using this infrastructure we were able to generate mutants at random. When choosing a mutant we used a non-uniform random distribution—the likelihood

[7] This weakness seems inherent to ranking methods based on black-box testing.

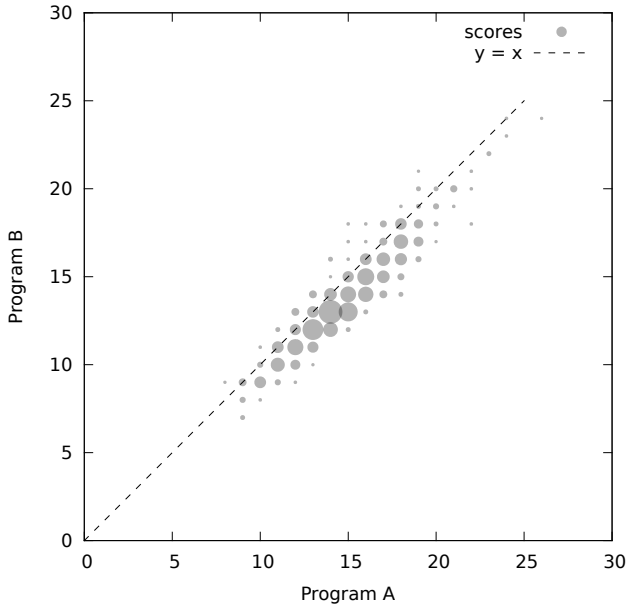

Figure 3.7: Scores of programs from pair 1. Circles lying on the dashed diagonal line represent runs where scores of both programs were equal.

of choosing a faulty version of code at a given location was 5 times smaller than leaving the code unchanged. We did this to keep the average number of mutations in the generated programs down; otherwise most of them would fail on all inputs and the rankings would become trivial.

The specific test we applied was to look at the relative stability of the scores for two randomly chosen programs. We first picked the two mutated programs, at random, and then ranked them 500 times using our method. Each of the 500 rankings involved 30 other mutants, each time chosen at random. The results for one such pair of programs are shown in the Figure 3.7. Each circle in the graph corresponds to a pair of scores received by the two selected programs in a run. When the same pair of scores occurs many times the circle is larger; the area of the circle is proportional to the number of times that pair occurs.

An interesting feature of the graph is the line '$y = x$': any points lying on this line represent rankings where both programs got the same score. Points lying below the line are rankings where program A got a higher score, and points above the line are ones where program B got the higher score.

The scores of both programs in the graph are quite variable, with most of the scores falling between 10 and 20. However, when ranking programs we are mostly interested in the relative difference between two programs; these are much less dispersed. Program A, whose scores are on the x axis, is typically better than program B by 0 to 2 points. Also, the vast majority of points lie

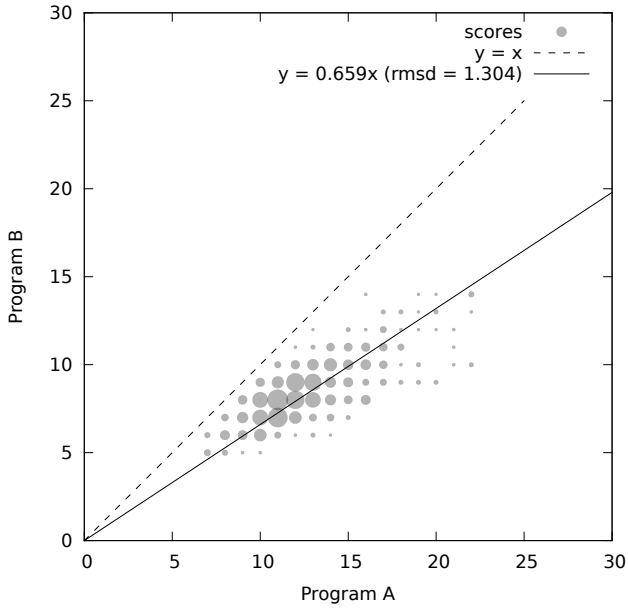

Figure 3.8: Scores of programs from pair 2.

on or below the '$y = x$' line, indicating that the ranking algorithm normally ranks program A higher than program B. (The few points lying above this line are drawn with small circles, indicating that not many rankings gave those scores.)

Further examination reveals that A contains one mutation and B contains the same one and two more. Despite that, due to some interaction between bugs, there are tests that fail for A and succeed for B. (We explore these interactions in Section 5.3.) Still, our method identifies A as the more correct program most of the time.

The results for another pair are shown in Figure 3.8. The two programs contain unrelated mutations and, according to our method, the mutations in program B are more serious than the ones in A. This is indicated by the fact that all circles are below the dashed diagonal line. Again the variability of scores is quite high, but the differences remain within a small range. We arrive at even smaller differences when we look at the ratio of one score to the other. The solid line crossing the origin on the graph is fitted to the pairs of scores with slope 0.659 and error of 1.304, indicating that the score of one program is usually roughly 0.659 times the score of the other.

Most graphs that we saw looked like these two with the first type being more common, even when programs had unrelated mutations. The fact that the differences in scores are not so variable is favourable to our method, but we were curious about the high spread of scores. It is explained by the fact that

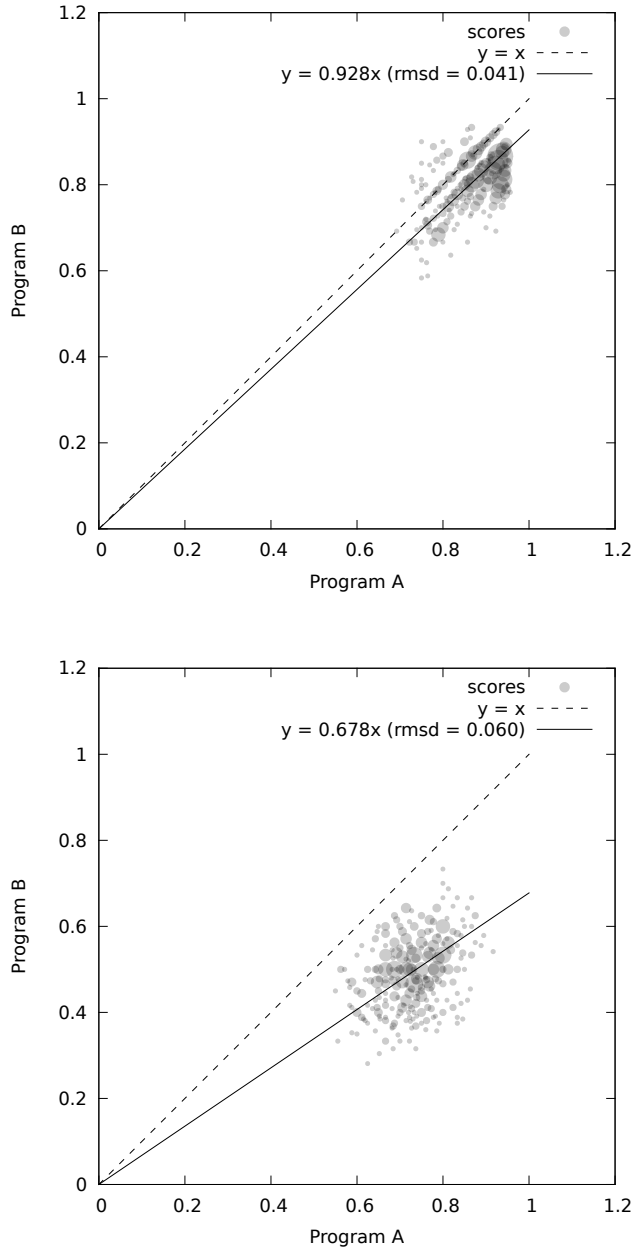

Figure 3.9: Normalized scores of programs from pairs 1 and 2.

in each run, depending on the other 30 programs chosen, the total number of inferred bugs differs. Thus, in each run the maximum score might be different from previous runs. To counter this effect, we normalized the scores, dividing each of them by the maximum score seen for a given set of runs so that the score for a program is in effect the percentage of tests that it passed.

Figure 3.9 contains graphs for the same pairs of programs as Figures 3.7 and 3.8 but with normalized scores. As we can see in the first graph both programs were getting good scores and in most cases program A was winning by a small margin. The solid line is fitted to the results with an error of about 4%. Normalized scores in the second graph are more varied, concentrated around 73% for A and 50% for B. A line fitted, starting at the origin, has an error of 6%.

We found that all pairs of programs examined had spreads of this magnitude and in almost all cases the line was fitted with an error of below 6%.

5.3 Inferred "bugs" = real bugs?

Aside from mathematical properties such as stability, we would also like to know that the ranking produced by our algorithm corresponds to what we intuitively expect.

What would like to do is take a well-understood set of programs, where we know what the bugs are, and see if our method infers a similar set of bugs to us. Just as before, since we don't really have such a set of programs, we generate programs by mutating the solution to the email anonymiser problem in various ways chosen by us; we assume that each mutation introduces one bug into the program. Program mutation was described in more detail in Section 5.2.

This time, we picked just four allowable mutations in the email anonymiser, compared with 18 mutation points in the stability testing. The reasons for this are twofold:

- By picking only four mutations, we were able to try to find four mutations that introduce independent faults into the program. If the faults are not independent then it's not clear that our ranking algorithm should treat each fault as a separate bug.
- Instead of generating a random set of mutants, as in the stability testing, we can instead run the ranking algorithm on *all* mutants at the same time. This should make our conclusions more reliable.

Table 3.1 shows the mutations we chose and the fault each introduces.

Given the four mutations in Table 3.1, there are 16 mutated programs. We use our algorithm to rank all these programs against each other, together with a 17th program that always crashes on any input.

We give the mutated programs names, from A0 to A16. A0 is the program that always crashes, the buggiest program; Table 3.2 shows which mutations the other programs have.

The ranking algorithm produces a partial order of the programs given by how correct they are. As we did in Section 4.1, we can visualise this ranking as

Name	Meaning
"_"	All alphanumeric characters in the email address are replaced by an underscore, including the first one: `kalle@anka.se` becomes `_____@____.__` instead of `k____@a___.s_`
"@"	An email address may contain several @-signs (this is disallowed by the specification): `kalle@anka@se` becomes `k____@a___@s_`
take 1	Some parts of the text are discarded: `kalle@anka.se` becomes `k____a___s_`
"."	A dot isn't considered an email address character: `kalle@anka.se` becomes `k____@a___.se`

Table 3.1: The mutations we made to the email anonymizer.

Mutation	A1	A2	A3	A4	A5	A6	A7	A8
take 1		✓		✓		✓		✓
"."			✓	✓			✓	✓
"@"					✓	✓	✓	✓
"_"								
Mutation	A9	A10	A11	A12	A13	A14	A15	A16
take 1		✓		✓		✓		✓
"."			✓	✓			✓	✓
"@"					✓	✓	✓	✓
"_"	✓	✓	✓	✓	✓	✓	✓	✓

Table 3.2: The mutations that each program has.

a graph of programs, where we draw an edge from one program to another if the first program passes all the testcases that the second one does. This visualisation for our set of 17 programs is given in Figure 3.10, where A1 (at the top) passes all testcases and A0 (at the bottom) passes none.

Does this graph look how we expect it to? To answer this, we should decide what we expect the graph to look like. As explained above, we hoped that all four mutations introduced independent bugs, in which case our algorithm would produce the graph shown in Figure 3.11. This graph shows the programs ordered by what mutations they have, rather than what bugs are inferred for them. This is the ideal graph we would like to get when we rank the 17 programs; any difference between our actual ranking and this graph is something we must explain.

The ranking we actually get, in Figure 3.10, is actually rather different! The best we can say is that all programs are at about the same "level" in the ideal graph and the actual one, but the structure of the two graphs is quite different.

If we look closer at the ranking, we can see that four pairs of programs are ranked equal: A2 and A10, A4 and A12, A6 and A14, and A8 and A16. These programs have different mutations but apparently the same behaviour. There is a pattern here: one program in each pair has the "take 1" mutation, the

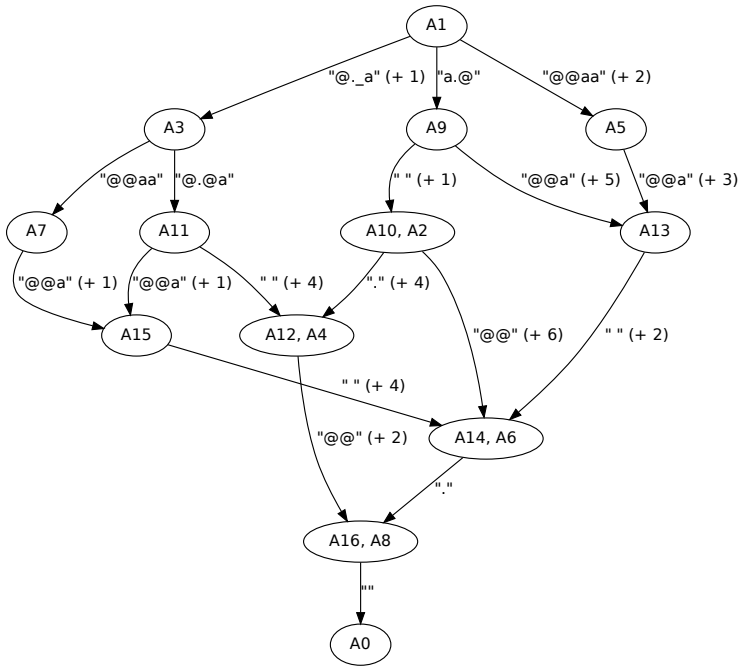

Figure 3.10: The mutants' ranking

other has both "take 1" and "_". What happens is that the "take 1" mutation subsumes the "_" error: a program with either mutation censors all email addresses incorrectly, but a program with the "take 1" mutation also mutilates text outside of email addresses.

So the mutations are not independent bugs as we hoped, which explains why we don't get the graph of Figure 3.11. Let us try to work around that problem. By taking that figure and merging any programs we know to be equivalent we get a different graph, shown in Figure 3.12. This is the graph we would get if all mutations were independent bugs, except for the matter of "take 1" subsuming "_".

This new graph is very close to the one our ranking algorithm produced, a good sign. However, there are still some differences we have to account for:

- **A5 ought to be strictly less buggy than A7 but isn't.** A5 has only the "@" mutation, while A7 has the "@" mutation and the "." mutation, so we would expect it to be buggier. However, this is not the case. The "@" mutation causes A5 to incorrectly see the string `a@b@c.de` as an email address (it isn't as it has two `@`-signs) and censor it to `a@b@c.d_`. A7, however, gives the correct answer here: because of the "."-mutation,

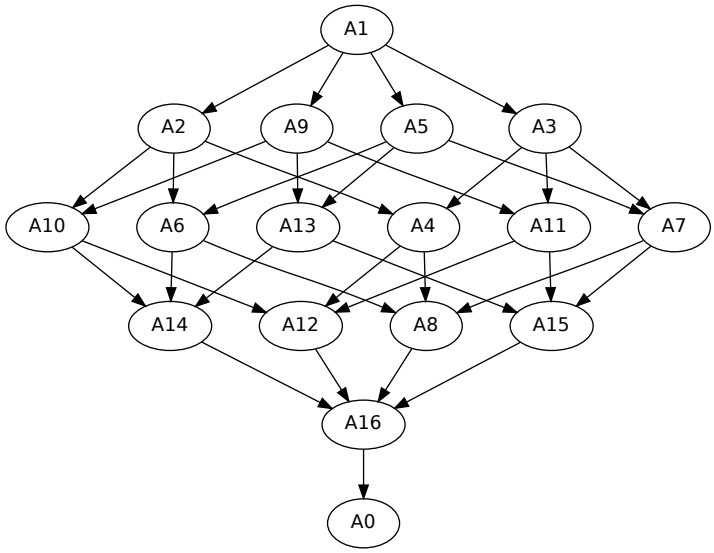

Figure 3.11: The mutants ranked by mutations, not by bugs

it thinks that the `.de` is not part of the email address and leaves it alone.

So this is a case where two bugs "cancel each other out", at least partially, and our algorithm does not try to detect this.

There is the same problem with programs A13 and A15, and it has the same cause, that the "." bug and the "@" bug interfere.

- **A6 is strictly buggier than A7 according to the algorithm, but we expect them to be incomparable.** A6 has the "@" and "take 1" mutations, while A7 has the "@" and "." mutations. We would expect A6 and A7 to have an overlapping set of bugs, then, but actually A7 passes every test that A6 does.

  This is because A6 actually censors every email address (any text containing an @-sign) incorrectly because of the "take 1" mutation. A7's mutation only affects email addresses with dots in, which A6 will fail on anyway because they will contain an @-sign.

  (Why, then, doesn't the "take 1" bug on its own subsume "."? The counterexample that our algorithm found is the test case `@.aa@`. This isn't an email address, since it has two @-signs, and a program with just the "take 1" mutation will behave correctly on it. However, a program with the "." mutation will identify the text as two email addresses, `@` and `aa@`, separated by a dot, and proceed to censor them to get `@.a_@`. Only if an email address may contain two @-signs does "take 1" subsume ".".)

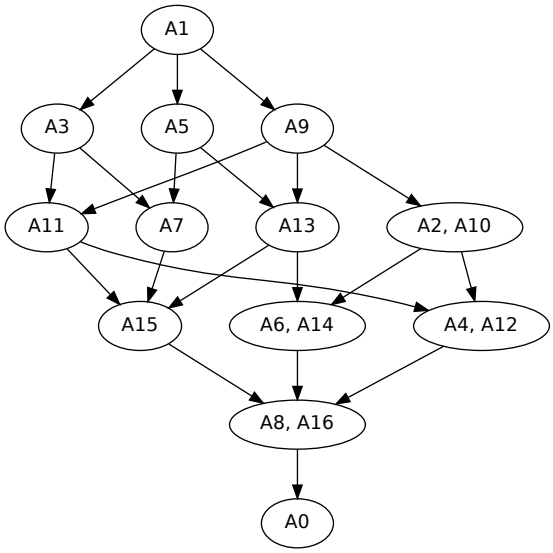

Figure 3.12: Figure 3.11 with some anomalous nodes merged

The same problem occurs with A14 and A15.

Overall, our ranking algorithm ranks the programs more or less according to which mutations they have, as we expected. There were some discrepancies, which happened when one bug subsumed another or two bugs "interfered" so that by adding a second bug to a program it became better. On the other hand, these discrepancies did not ruin the ranking and whenever bugs were independent, the algorithm did the right thing.

Our bugs interacted in several ways despite our best efforts to choose four independent bugs. We speculate that this is because the email anonymiser consists of only one function, and that a larger API, where the correctness of one function doesn't affect the correctness of the others, would be better-behaved in this respect.

5.4 Conclusion on Stability

We checked three important properties that we expect from a ranking method. Firstly, we concluded that the results are stable even when the test generator omits some of the relevant test cases. Secondly, we showed that when different subsets of "reasonable" programs are present this does not change the results of the ranking very much. And thirdly, we were able to explain the irregularities in a graph produced by our ranking by interactions between different faults.

It is impossible to provide a complete specification for the ranking method without refering to the way it works under the hood, however the three properties that we checked provide a reasonable *partial specification* that our method satisfies. The tests we applied increased our confidence in the ranking method, however one could imagine a more convincing variant of the last test where we use a large set of real programs to validate the stability of rankings.

6 Related Work

Much work has been devoted to finding representative test-suites that would be able to uncover all bugs even when exhaustive testing is not possible. When it is possible to divide the test space into partitions and assert that any fault in the program will cause one partition to fail completely it is enough select only a single test case from each partition to provoke all bugs. The approach was pioneered by Goodenough and Gerhart [1975] who looked both at specifications and the control structure of tested programs and came up with test suites that would exercise all possible combinations of execution conditions. Weyuker and Ostrand [1980] attempted to obtain good test-suites by looking at execution paths that they expect to appear in an implementation based on the specification. These methods use other information to construct test partitions, whereas our approach is to find the partitions by finding faults in random testing.

Lately, test-driven development has gained in popularity, and in a controlled experiment from 2005 [Erdogmus et al., 2005] Erdogmus et. al. compare its effectiveness with a traditional test-after approach. The result was that the group using TDD wrote more test cases, and tended to be more productive. These results are inspiring, and the aim with our experiment was to show that property-based testing (using QuickCheck) is a good way of conducting tests in a development process.

In the design of the experiments we were guided by several texts on empirical research in software engineering, amongst which [Basili et al., 1986, Kitchenham et al., 2002, Wohlin et al., 2000] were the most helpful.

7 Conclusions

We have designed an experiment to compare property-based testing and conventional unit testing, and as part of the design we have developed an unbiased way to assess the "bugginess" of submitted solutions. We have carried out the experiment on a small-scale, and verified that our assessment method can make fine distinctions between buggy solutions, and generates useful results. Our experiment was too small to yield a conclusive answer to the question it was designed to test. In one case, the interval sets, we observed that all the QuickCheck test suites (when they were written) were more effective at detecting errors than any of the HUnit test suites. Our automated analysis suggests, but does not prove, that in one of our examples, the code developed using QuickCheck was less buggy than code developed using HUnit. Finally, we observed that QuickCheck users are less likely to write test code than HUnit

users—even in a study of automated testing—suggesting perhaps that HUnit is easier to use.

The main weakness of our experiment (apart from the small number of subjects) is that students did not have enough time to complete their answers to their own satisfaction. We saw this especially in the cryptarithm example, where more than half the students submitted solutions that passed no tests at all. In particular, students did not have time to complete a test suite to their own satisfaction. We imposed a hard deadline on students so that development time would not be a variable. In retrospect this was probably a mistake: next time we will allow students to submit when they feel ready, and measure development time as well.

In conclusion, our results are encouraging and suggest that a larger experiment might demonstrate interesting differences in power between the two approaches to testing. We look forward to holding such an experiment in the future.

Acknowledgements

This research was sponsored by EU FP7 Collaborative project *ProTest*, grant number 215868.

Bibliography

The Cover project.
http://web.archive.org/web/20070114174208/http://www.coverproject.org/.

→ 1 citation on page: 2

Integration of QuickCheck and McErlang.
https://babel.ls.fi.upm.es/trac/McErlang/wiki/QuickCheck/McErlang.
→ 1 citation on page: 2

J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.
→ 3 citations on 2 pages: 50 and 52

C. Artho, K. Havelund, and S. Honiden. Visualization of concurrent program executions. In *COMPSAC '07: Proc. of the 31st Annual International Computer Software and Applications Conference*, pages 541–546, Washington, DC, USA, 2007. IEEE Computer Society.
→ 1 citation on page: 75

T. Arts and L.-Å. Fredlund. Trace analysis of Erlang programs. *SIGPLAN Notices*, 37(12):18–24, 2002.
→ 1 citation on page: 75

T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with quviq QuickCheck. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1. doi: http://doi.acm.org/10.1145/1159789.1159792.
→ 5 citations on 5 pages: 7, 47, 51, 53, and 79

V. R. Basili, R. W. Selby, and D. H. Hutchens. Experimentation in software engineering. *IEEE Trans. Softw. Eng.*, 12(7):733–743, 1986. ISSN 0098-5589.
→ 2 citations on 2 pages: 81 and 103

M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180, New York, NY, USA, 2006. ACM. ISBN 1-59593-263-1. doi: http://doi.acm.org/10.1145/1146238.1146258.

→ 1 citation on page: 46

F. W. Burton. An efficient functional implementation of FIFO queues. *Inf. Proc. Lett*, (14):205–206, 1982.
→ 1 citation on page: 14

G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. A. Visaggio. Evaluating advantages of test driven development: a controlled experiment with professionals. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 364–371, New York, NY, USA, 2006. ACM. ISBN 1-59593-218-6. doi: http://doi.acm.org/10.1145/1159733.1159788.
→ 1 citation on page: 9

R. Carlsson and D. Gudmundsson. The new array module. In B. Däcker, editor, *13th International Erlang/OTP User Conference*, Stockholm, 2007. Available from `http://www.erlang.se/euc/07/`.
→ 1 citation on page: 15

I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 84–94, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-734-6. doi: http://doi.acm.org/10.1145/1273463.1273476. URL http://doi.acm.org/10.1145/1273463.1273476.
→ 1 citation on page: 3

K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: http://doi.acm.org/10.1145/351240.351266.
→ 7 citations on 7 pages: 2, 12, 34, 47, 51, 53, and 79

K. Claessen and J. Hughes. Testing monadic code with QuickCheck. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 65–77, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6. doi: http://doi.acm.org/10.1145/581690.581696.
→ 1 citation on page: 47

M. Cronqvist. Troubleshooting a large Erlang system. In *ERLANG '04: Proc. of the 2004 ACM SIGPLAN workshop on Erlang*, pages 11–15, New York, NY, USA, 2004. ACM.
→ 1 citation on page: 51

S. Dehnadi and R. Bornat. The camel has two humps. www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf, 2006.
→ 1 citation on page: 93

H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software*

Engineering, 31:226–237, 2005. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2005.37.
→ 1 citation on page: 103

M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2007.01.015.
→ 1 citation on page: 46

L.-Å. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. *SIGPLAN Not.*, 42(9):125–136, 2007.
→ 2 citations on 2 pages: 8 and 73

E. R. Gansner and S. C. North. An open graph visualization system and its applications. *Software - Practice and Experience*, 30:1203–1233, 1999.
→ 1 citation on page: 65

J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, New York, NY, USA, 1975. ACM. doi: http://doi.acm.org/10.1145/800027.808473.
→ 1 citation on page: 103

D. Hamlet. When only random testing will do. In *Proceedings of the 1st international workshop on Random testing*, RT '06, pages 1–9, New York, NY, USA, 2006. ACM. ISBN 1-59593-457-X. doi: http://doi.acm.org/10.1145/1145735.1145737. URL http://doi.acm.org/10.1145/1145735.1145737.
→ 1 citation on page: 3

J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for Java container classes. *IEEE Trans. Software Eng.*, 33(8):526–543, 2007.
→ 3 citations on 3 pages: 5, 18, and 45

D. Herington. HUnit: A unit testing framework for Haskell. http://hackage.haskell.org/package/HUnit-1.2.2.1, January 2010.
→ 1 citation on page: 80

M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *POPL '87: Proc. of the 14th ACM SIGACT-SIGPLAN symposium on Principles of Prog. Lang.*, pages 13–26, New York, NY, USA, 1987. ACM.
→ 1 citation on page: 56

C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1: 271–281, 1972.
→ 1 citation on page: 37

J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2): 98–107, 1989.
→ 1 citation on page: 4

J. Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925, 1995.
→ 1 citation on page: 4

J. Hughes. QuickCheck testing for fun and profit. In M. Hanus, editor, *PADL*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2007. ISBN 978-3-540-69608-7.
→ 3 citations on 3 pages: 47, 51, and 53

J. Hughes, U. Norell, and J. Sautret. Using temporal relations to specify and test an instant messaging server. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 95–102, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-970-1. doi: http://doi.acm.org/10.1145/1808266.1808281. URL http://doi.acm.org/10.1145/1808266.1808281.
→ 1 citation on page: 2

D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *In Proc. of the 19th International Conference on Software Engineering*, pages 360–370, 1997.
→ 1 citation on page: 75

JUnit.org. JUnit.org resources for test driven development. http://www.junit.org/, January 2010.
→ 1 citation on page: 80

B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28:721–734, 2002. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2002.1027796.
→ 1 citation on page: 103

Klein, Lu, and Netzer. Detecting race conditions in parallel programs that use semaphores. *Algorithmica*, 35:321–345, 2003.
→ 1 citation on page: 73

L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
→ 1 citation on page: 67

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.
→ 2 citations on 2 pages: 7 and 56

S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGARCH Comput. Archit. News*, 36(1):329–339, 2008.
→ 2 citations on 2 pages: 57 and 73

S. Maoz, A. Kleinbort, and D. Harel. Towards trace visualization and exploration for reactive systems. In *VLHCC '07: Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 153–156, Washington, DC, USA, 2007. IEEE Computer Society.
→ 1 citation on page: 75

B. Marick. How to Misuse Code Coverage, 1999. URL http://www.exampler.com/testing-com/writings/coverage.pdf.
→ 1 citation on page: 1

R. L. McCasland and A. Bundy. MATHsAiD: a mathematical theorem discovery tool. In *Proceedings of the Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'06)*, 2006.
→ 1 citation on page: 47

S. Muggleton and L. de Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19:629–679, 1994.
→ 1 citation on page: 46

M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.
→ 2 citations on 2 pages: 8 and 75

R. H. B. Netzer and B. P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *In Proc. of the 1990 Int. Conf. on Parallel Processing*, pages 93–97, 1990.
→ 1 citation on page: 73

R. Nieuwenhuis and A. Oliveras. Proof-producing congruence closure. In *RTA '05*, pages 453–468, Nara, Japan, 2005. Springer LNCS.
→ 1 citation on page: 23

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. ISBN 0-521-66350-4.
→ 1 citation on page: 35

C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT '08/FSE-16: Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145, New York, NY, USA, 2008. ACM.
→ 1 citation on page: 74

G. Plotkin and J. Power. Notions of computation determine monads. In *Proc. FOSSACS 2002, Lecture Notes in Computer Science 2303*, pages 342–356. Springer, 2002.
→ 1 citation on page: 6

K. Sen. Race directed random testing of concurrent programs. *SIGPLAN Not.*, 43(6):11–21, 2008a.
→ 1 citation on page: 74

K. Sen. Race directed random testing of concurrent programs. *SIGPLAN Not.*, 43:11–21, June 2008b. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1379022.1375584. URL http://doi.acm.org/10.1145/1379022.1375584.
→ 1 citation on page: 8

H. Svensson and L.-Å. Fredlund. A more accurate semantics for distributed Erlang. In *Erlang '07: Proc. of the 2007 SIGPLAN Erlang Workshop*, pages 43–54, New York, NY, USA, 2007. ACM.
→ 1 citation on page: 61

G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.
→ 1 citation on page: 1

The GHC Team. The Glasgow Haskell Compiler. http://www.haskell.org/ghc/, January 2010.
→ 1 citation on page: 83

N. Tillmann and J. de Halleux. Pex—white box test generation for .NET. In *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-79123-2.
→ 1 citation on page: 79

N. Tillmann and W. Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, 2005. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1095430.1081749.
→ 1 citation on page: 79

B. Topol, J. Stasko, and V. Sunderam. Integrating visualization support into distributed computing systems. *Proc. of the 15th Int. Conf. on: Distributed Computing Systems*, pages 19–26, May-Jun 1995.
→ 1 citation on page: 75

E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Trans. Softw. Eng.*, 6(3): 236–246, 1980. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1980.234485.
→ 1 citation on page: 103

U. T. Wiger. Extended process registry for Erlang. In *ERLANG '07: Proc. of the 2007 SIGPLAN workshop on ERLANG Workshop*, pages 1–10, New York, NY, USA, 2007. ACM.
→ 1 citation on page: 51

C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-8682-5.
→ 1 citation on page: 103

A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, 2002. ACM. ISBN 1-58113-514-9. doi: http://doi.acm.org/10.1145/587051.587053.
→ 1 citation on page: 79